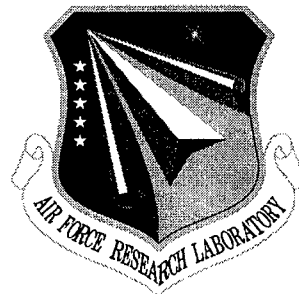AFRL-IF-RS-TR-2001-131
Final Technical Report
June 2001

KNOWLEDGE ACQUISITION FOR LARGE KNOWLEDGE BASES: INTEGRATING PROBLEM-SOLVING METHODS AND ONTOLOGIES INTO APPLICATIONS

USC Information Sciences Institute

20010907 144

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**AIR FORCE RESEARCH LABORATORY
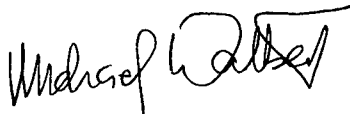INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-131 has been reviewed and is approved for publication.

APPROVED: RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: MICHAEL TALBERT, Technical Advisor
Information Technology Division
Information Directorate

KNOWLEDGE ACQUISITION FOR LARGE KNOWLEDGE BASES: INTEGRATING PROBLEM-SOLVING METHODS & ONTOLOGIES INTO APPLICATIONS

Yolanda Gil

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jun 01	Final Apr 97 - Sep 00

4. TITLE AND SUBTITLE
KNOWLEDGE ACQUISITION FOR LARGE KNOWLEDGE BASES: INTEGRATING PROBLEM-SOLVING METHODS AND ONTOLOGIES INTO APPLICATIONS

5. FUNDING NUMBERS
G - F30602-97-1-0195
PE - 62301E
PR - IIST
TA - 00
WU - 04

6. AUTHOR(S)

Yolanda Gil

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency   AFRL/IFTD
3701 North Fairfax Drive   525 Brooks Rd
Arlington, VA 22203-1714   Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2001-131

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi, IFTD, 315-330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The HPKB initiative seeks to develop large, reusable libraries of ontologies and problem- solving methods which will ease the construction and maintenance of large knowledge based systems. A critical fact of HPKB is to understand how these ontologies and problem-solving methods can be brought together to produce applications and to develop tools and techniques that support that process.

14. SUBJECT TERMS
Expect, Plan Representation and Reasoning, Deriving Expectations to Guide Knowledge Base Creation, Extending the Role Limiting Approach, User Studies of Knowledge Acquisition Tools, Evaluations and End Users

15. NUMBER OF PAGES
128

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Outline

Outline of Figures

1. Introduction

The HPKB initiative seeks to develop large, reusable libraries of ontologies and problem solving methods which will ease the construction and maintenance of large knowledge based systems. A critical facet of HPKB is to understand how these ontologies and problem solving methods can be brought together to produce applications and to develop tools and techniques that support that process.

When the ontologies and problem solving methods are brought together, dependencies will be set up between them. If one problem solving method is chosen, certain parts of an ontology will be used, while others will not be needed. On the other hand, if a different method is chosen, different parts of the ontology will be required. Capturing these interdependencies that arise when ontologies and problem solving methods are actually used in building a KBS is critical to providing intelligent support for maintenance, evolution and knowledge acquisition. Our work focuses on capturing these interdependencies and developing tools that use them to support system builders and domain experts as they bring together the knowledge in the libraries and make it operational in knowledge based applications.

Consider how this *knowledge of how knowledge is used* could be employed: if the shared ontology a KBS is based on changes after the system is built, the system builder should be notified, but only of the changes that affect his system. This can only be done if one understands how the ontology is used in that particular system. Similarly, in knowledge acquisition, when a user adds new knowledge to a KBS, he needs to be prompted for any additional knowledge that may be required to make use of that new information, but no extraneous knowledge should be requested. Both of these capabilities depend on understanding the tie between problem solving methods and ontologies: understanding how the problem solving methods make use of the ontologies and domain knowledge, and what domain knowledge is required to support problem solving.

These ideas are key to our research within the EXPECT architecture investigated in the work reported here. EXPECT's architecture includes a knowledge representation and reasoning system, knowledge analysis and interaction dialogue capabilities, and knowledge acquisition tools and interfaces.

1.1 Knowledge Acquisition

Knowledge acquisition tools have expectations about the kind of knowledge that needs to be added to a system to support reasoning, and they use these expectations to guide acquisition. Most successful knowledge acquisition tools create these expectations using what is known as a role-limiting approach. Role -limiting approaches center knowledge base construction on filling the roles that domain -dependent knowledge plays in a domain-independent problem -solving method. For example, a knowledge acquisition tool for a heuristic classification problem -solving method expects that the user will provide knowledge about classes and features that are used in classification. The same general-purpose problem-solving method can be used in many applications that use that problem-solving approach, and users interact with the tool to fill in the roles in that problem-solving method. In our example, for a medical diagnosis application the classes

1

would be diseases and features would be the data about the patient, while for a computer diagnosis system the classes would be computer failures and the features would be the symptoms of the computer. Examples of such tools include MORE [Eshelman 1988], SALT [Marcus 1988], and ROGET [Bennett 1985]. Because these tools understood how knowledge would be used in problem solving, they could provide much more guidance in helping the user formulate the knowledge he was adding to a system correctly. However, the problem solving method is built into th e knowledge acquisition tool itself. Thus, when one selects a tool, one also determines the problem -solving method that is employed by the application. The diffi culty is that most realistically-sized knowledge-based systems require an assortment of prob lem-solving methods—a tool that uses a single method is of limited utility. In addition, many knowledge acquisition tools need to be built, one for each problem -solving method. For these reasons, these approaches to knowledge acquisition have limited applicability [Musen 1992].

Our work overcomes these problems with a com plementary approach to knowledge acquisition. In order to support flexible tools, EXPECT has an explicit representation of all the kinds of knowledge (both factual and procedural) in volved in a task. EXPECT's representation of domain facts and problem -solving methods will be tightly integrated with LOOM, an advanced knowledge -representation system. As a result, the problem - solving methods are not treated as black boxes, but instead t heir definitions are represented in a language that the system understands and about which the system can reason. EXPECT 's knowledge-acquisition tools will be able to support users in changing these problem -solving methods, defining new ones, and composin g them together to configure an overall problem -solving method for an application. Building a new application with EXPECT, however, would require developers to define by hand all the problem-solving methods that their applications might need. EXPECT itsel f will not support the reuse of problem-solving methods from one application to another.

1.2 EXPECT

The main goal of the EXPECT project [Swartout and Gil 1995, Gil and Paris 191994, Neches et al. 1985, Swartout et al. 1991] is to make knowledge -based systems more accessible to end users. A decade of work on the Explainable Expert Systems (EES) project at ISI lead to the development of new approaches to natural language generation, dialogue-based explanations, and documentation [Swartout and Moore 1994, Swartout et al. 1991]. Recent work has concentrated on tools that support the acquisition of knowledge from users in terms they can relate to, highlighting domain knowledge instead of programming details [Gil and Melz 1996, Gil and Paris 1994, Gil 1994]. EXPECT's approach to modeling, representing, and using knowledge was shaped to provide appropriate support for explanation and knowledge acquisition tools. EXPECT provides a highly declarative framework for building knowledge-based systems which is based upon Loom [MacGregor 1988, MacGregor 1990], a state-of-the-art knowledge representation system of the KL-ONE family. This framework captures the design underlying a KBS, including how domain information is used in problem solving, and what factual knowledge is needed to support the KBS's problem solving methods. This design information is then used to guide knowledge acquisition.

To build a knowledge based system in EXPECT, one begins by representing general knowledge about a domain as Loom entities (e.g., conce pts, relations, and instances). General problem -solving strategies are represented in a procedural -style language that allows subgoal posting, control programming constructs, and expressive parameter typing. Using a form of partial evaluation, an automatic method instantiator then compiles this general knowledge into a domain -specific knowledge based system. This process is recorded in a design history that captures the interdependencies in the knowledge-based system, such as how factual knowledge is used i n problem solving. EXPECT's knowledge acquisition routines then use this information about dependencies to guide knowledge acquisition.

Knowledge acquisition tools support users by having expectations about the kinds of knowledge that a user may want to provide to a system. EXPECT uses its design record to form expectations about what knowledge it needs to acquire based on the current contents of its knowledge bases. By using the design record, EXPECT understands how the factual knowledge is used in the problem-solving methods. Thus, EXPECT can guide users to provide enough information about instances and concepts to ensure that problem solving can be carried out. Because it understands how the problem -solving methods achieve the task goals, EXPECT can ens ure that there are methods to achieve the goals that arise during problem solving. If the user changes a method definition or a factual description, the expectations are rederived. Understanding the interactions among the different pieces of knowledge is also useful to propagate the effects of a local change made by the user. This allows the system to detect inconsistencies as well as the need to request additional knowledge from the user. EXPECT also includes an agenda mechanism to handle requests for user interventions, providing users with an abstract view of the knowledge acquisition tasks that they need to attend to.

EXPECT provides a flexible approach to knowledge acquisition because 1) it allows users to define arbitrary problem-solving methods for a task, 2) the same tool can be reused for many different applications to accomplish diverse types of tasks, and 3) its expectations are derived according to the knowledge that is available about the task. In other approaches, knowledge acquisition tools are built for specific types of tasks, and the expectations are predesigned based on the task definition and hard -coded into the tool. This approach is not practical in real applications because it is hard to determine beforehand the type of problem-solving method that is needed for a new application and because applications often do not conform exactly to the type of task that a tool was designed for.

1.3 Overview

This report summarizes the extensions done to EXPECT under the High Performance Knowledge Bases Program. The work is described in this report organized in the following topics and chapters:

- The current EXPECT architecture and the features that it includes that are crucial to support problem solving and knowledge acquisition work.

- The ontologies fo r plan representation and reasoning that were developed, which were used in the HPKB Challenge Problems.
- A knowledge acquisition tool to acquire problem solving knowledge
- A reusable problem solving method for plan evaluation that was also used to support knowledge acquisition from users
- A methodology to conduct user evaluations of knowledge acquisition tools
- A report on user evaluations of Army officers conducted at the Army Battle Command Battle Lab in Ft. Leavenworth, KS in August 1999 as part of the HPKB Knowledge Acquisition Critical Component Experiment.

The next six chapters in this report address each of these topics in more depth.

2. EXPECT: A User-Centered Environment for Developing Knowledge-Based Systems

Although knowledge based systems have been successfully deployed in many areas, a big impediment to more extensive use is the knowledge acquisition problem. Building a conventional knowledge based system is an expensive and time consuming process because both domain experts and AI experts mus t work together to capture the needed knowledge and represent it in a system. The EXPECT project has addressed this problem by creating a framework for building knowledge based systems that will empower domain experts to add knowledge to a knowledge based system themselves, while freeing them from the need to understand the details of the implementation or how the system is organized. During the past year, we have developed and implemented this knowledge acquisition facility, and have demonstrated it in t he context of a system to evaluate military transportation plans.

Unlike conventional acquisition tools which use a fixed set of constraints built into the tool to guide knowledge acquisition, EXPECT analyzes the design and structure of a knowledge based system to guide acquisition. EXPECT is thus more flexible than conventional tools and can be applied to a broader range of problems. The key to EXPECT's approach to knowledge acquisition is that the EXPECT framework captures critical aspects of the design of a knowledge based system. EXPECT knows how various kinds of knowledge in a system interact, such as how factual knowledge is used by problem solving methods, and what factual knowledge is needed to support reasoning. Because EXPECT understands these interactions, it can guide a user in adding new knowledge to a system and prompt the user for additional information if it is needed. EXPECT currently supports the addition and modification of both factual knowledge and problem solving methods. By understanding more of the design of a knowledge based system, EXPECT makes it possible for less computer -knowledgeable users to add knowledge to a system successfully.

2.1. Introduction

Our goal is to construct a framework for building knowledge -based systems (KBSs) that can help users make modifications. The premise underlying this goal is that the framework understands a lot of the systems' design and their structure—how they solve problems, what knowledge is needed to support problem solving, and what assumptions they make. This knowledge can be exploited to make the system understandable and to guide users in making modifications.

EXPECT is the framework for knowledge based systems that we are developing to support knowledge acquisition and explanation. A central idea behind EXPECT is the notion that more powerful acquisition and explanation tools can be constructed if acquisition and explanation concerns are reflected in the structure of the knowledge based systems we create. That is, rather than developing tools that operate on conventional knowledge based systems, it is first necessary to modify the architecture of the target knowledge based systems so that they will be structured in a way that provides better

support for knowledge acquisition and explanation. It is then possible to build to ols that exploit this additional information to provide enhanced capabilities.

In prior work on the EES framework [Neches et al, 1985; Swartout et al., 1991] we explored the architectural modifications that are needed to support explanation. EXPECT extends the EES framework to support knowledge acquisition. In this chapter, we discuss the architectural features that support knowledge acquisition.

There are several knowledge acquisition capabilities that we seek to support with the EXPECT architecture:

1. *Users should be able to modify both factual knowledge and problem solving knowledge.* Although many acquisition systems provide good support for modifying factual knowledge, support for modifying problem solving knowledge is more limited. In early acqui sition systems (such as MORE [Eshelman 1988], SALT [Marcus and McDermott 1989], and ROGET [Bennett 1985]), a single problem solving strategy (e.g. heuristic classification) was built into the tool. As a result, when one selected a tool, one also determined the problem -solving strategy. However, many realistically-sized knowledge-based systems use several problem solving strategies, so a tool that only supports a single strategy won't be much help. Additionally, if the user changes his mind about which problem-solving strategy is appropriate, he may also have to change tools, potentially losing a lot of work in re -configuring the domain knowledge. Recent knowledge acquisition work [Klinker et al. 1991; Musen and Tu 1993] has partially addressed these prob lems by creating tools that can use multiple problem -solving methods. In PROTÉGÉ II [Musen and Tu, 1993], problem solving methods are composed of pre -encoded building blocks. PROTÉGÉ II permits modific ations to problem solving by substituting one building block for another, however, users cannot modify the building blocks themselves. By constraining the user's options to just those that have been pre-encoded, he may not be able to make the modification he wants.

2. *Internal models (based on the knowledg e based system being modified) should guide knowledge acquisition rather than external ones (based on the acquisition tool).* Current acquisition systems cannot allow much modification to problem solving knowledge because they use external models of proble m solving that are built into the acquisition tool. By understanding what factual knowledge is needed to support problem solving the acquisition tool can form expectations about what additional knowledge is required when the user adds or modifies the syste m's know ledge base. In early acquisition tools, these expectations were built, by hand, into the tool itself. In more recent work, such as PROTÉGÉ II, the interdependencies between factual knowledge and problem solving building blocks are represented, but they are still entered by hand.

These limitations could be overcome and a wider range of modifications and problem solving methods could be supported if we could use an *internal* model for acquisition, that is, if the expectations for acquisition could be derived from the system itself. Furthermore, because the interdependencies would be derived from the system rather than entered by hand, they would change as the system changed, and would be more likely to be correct and consistent.

3. *KA tools should su pport modifications at a semantic (or knowledge) level rather than just at a syntactic level.* KA tools should help ensure that the knowledge a user adds makes sense, that is, that it is coherent and consistent with the rest of the knowledge base—not just that it is syntactically correct. In addition, we want to facilitate the addition of new knowledge by reducing the distance between what the user understands

6

and way the system represents it. The system must be able to represent and manipulate conceptual entities that are meaningful to users and that are used to describe the domain. To further facilitate acquisition, a KA tool should allow users to make modifications locally, and guide them in resolving the global implications of the changes. Providing assistance at the conceptual level and allowing gr eater flexibility in the use of terminology will free users to focus on what matters: getting the knowledge right.

This chapter describes several features of EXPECT'S architecture that directly support the acquisition goals we outlined above. Most of the e xamples in this chapter are based on a Course of Action evaluation tool, but some are from another domain we have used which is concerned with the diagnosis of faults in local area networks.

2.2. EXPECT Architecture Overview

A diagram of the overall EXPECT architecture appears in Figure 1. As we describe in the next section, EXPECT provides explicit and separate representations for different kinds of knowledge that go into a knowledge based system. EXPECT distinguishes domain facts, domain terminology and problem solving knowledge. These different sources of knowledge are integrated together by a program instantiator to produce a domain-specific knowledge based system. While the domain -specific system is being created, the program instantiator also cr eates a design history. The design history records the interdependencies among the different kinds of knowledge, such as what factual information is used by the problem solving methods. This i nformation is used by the knowledge acquisition routines to fo rm the expectations that guide the knowledge acquisition process.

7
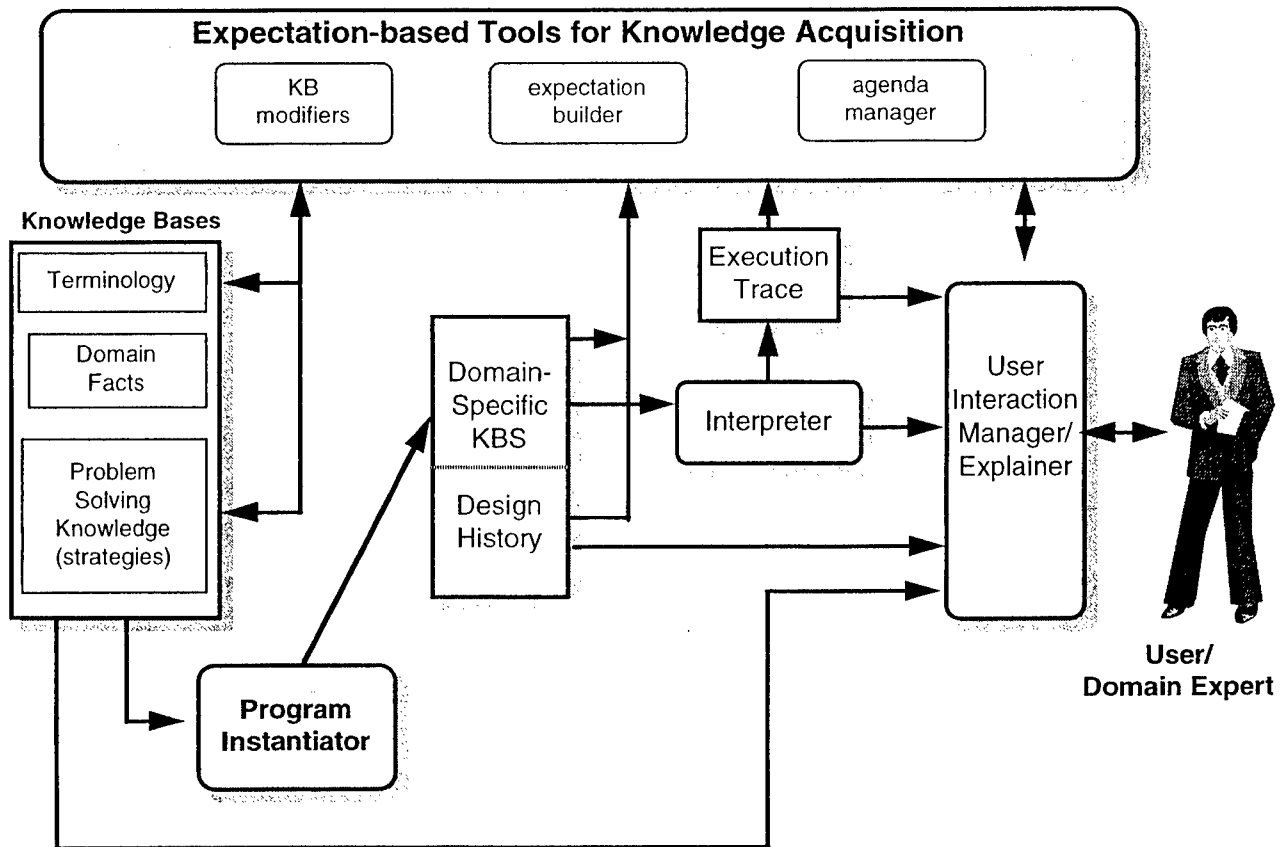
Figure 1. EXPECT Architecture

2.3. Representing Knowledge in EXPECT

EXPECT represents different types of knowledge separately and explicitly. After describing the kinds of knowledge that are represented in EXPECT we present in detail our approach to representing goals. This explicit representation of goals provides EXPECT with a better understanding of the problem solving process.

2.3.1 Separation of Knowledge

It is well established that a major source of difficulties in understanding, modifying and augmenting first generation knowledge based systems stemmed from the use of low - level knowledge representations that failed to distinguish different kinds of knowledge (see [Chandrasekaran and Mittal, 1982; Clancey, 1983b; Swartout, 1983]). In a first generation system, domain facts, problem solving know ledge, and terminological definitions were all expressed in rules. A single rule might mix together clauses concerned with the user int erface, the system's problem solving strategy and internal record-keeping. Because none of these different concerns were distinguished, it was often difficult to understand exactly what the rule was supposed to do, and when modifying a rule, it was difficult to see what the effect of the modification might be. Although early critiques of these representations focused on their failure to provide good explanations, these architectural flaws create problems for acquisition as well.

A number of second generati on expert system frameworks have emerged in recent years (see [Chandrasekaran, 1986; Clancey, 1983a; Hasling et al., 1984; Neches et al, 1985; Swartout, 1983; Swartout et al., 1991; Wielinga and Breuker, 1986]). A common theme among these frameworks is that they encourage a more abstract representation of domain knowledge and problem solving knowledge that makes distinctions between different kinds of knowledge explicit.

By moving toward architectures that allow a system builder to distinguish different kinds of knowledge and represent them separately and more abstractly, second generation frameworks increased the modularity of an expert system. This modularity facilitates KA by making systems easier to understand and augment.

In EXPECT, we distinguish three different kinds of knowledge: *domain facts*, *domain terminology*, and *problem solving knowledge*. Domain facts are the relevant facts about a system's domain. For example, the domain facts in a system for transportation planning might include the fact that the naval port of Los Angeles is Long Beach and that the maximum depth of Long Beach berths is 50 feet.

The domain terminology (or *ontology*) provides the conceptual structures that are used to describe a domain. In a transportation planning domain, the domain terminology would include concepts for various kinds of ports, such as airports and seaports, and concepts for describing the var ious kinds of movements [1] and materiel to be moved, among other things. Concepts can be defined in terms of other conc epts. Domain terminology provides a set of terms, or concepts, that can be used to describe some situation. The existence of a concept does *not* imply that the object it describes actually exists—concepts are just descriptions that may or may not apply in any given situation. Domain facts, on the other hand, use domain terminology to represent what exists.

To represent both domain facts and terminology, EXPECT uses Loom [MacGregor 1988]. Loom provides a descriptive logic representation language and a *classifier* for inference. Facts are represented as Loom *instances*, while terminology is represented using Loom *concepts*. Both instances and concepts are structured, frame -based representations with slots that indicate relations in which the object is involved.

Loom's classifier benefits knowledge acquisition. Given a set of defined concepts, the classifier can automatically organize them into an is -a (or subsumption) hierarchy by analyzing their definitions. For example, suppose we define the following two concepts:

```
an AIRLIFT-MOVEMENT is a kind of MOVEMENT whose destination is
    an AIRPORT
an EXPRESS-MOVEMENT is a kind of MOVEMENT whose duration is
    less than one DAY and whose destination is an AIRPORT
```

The classifier would figure out that an express -movement is also a kind of airlift-movement, since the definitions above state that any movement whose destin ation is an airport is an airlift-movement, and express-movement meets that criteria. This is shown in Figure 2.

---

[1] In transportation planning, a "movement" specifies what is to be moved from an origin to a destination at a particular time using some set of vehicles.

Before Classification   After Classification
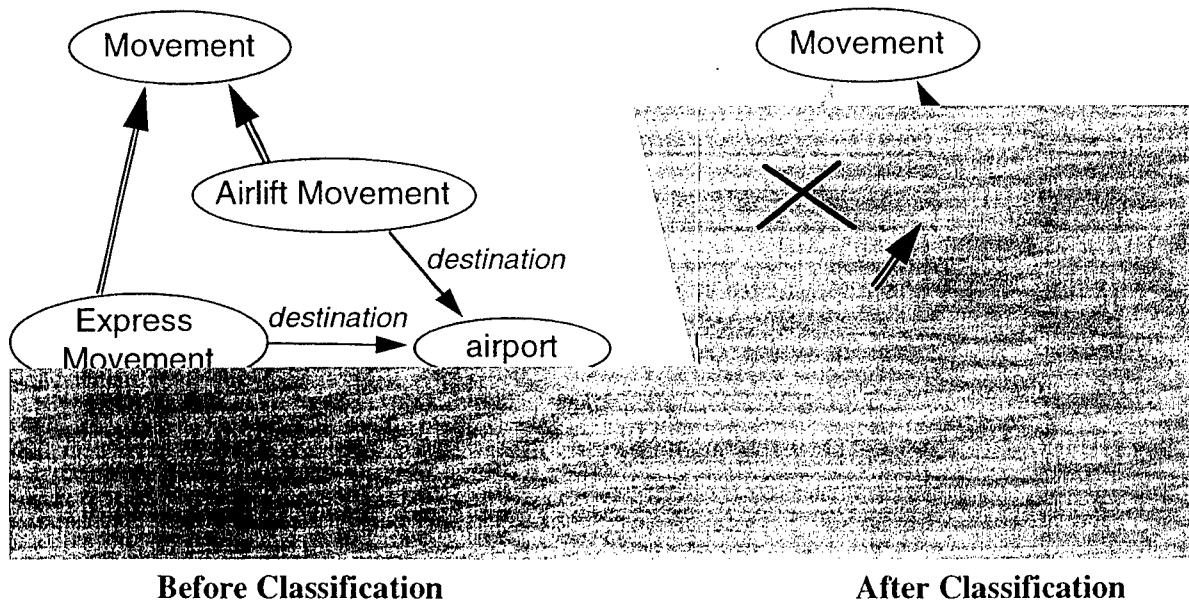
Figure 2. Loom Classifier Organizes Concept Hierarchy

Loom provides a declarative foundation for representing concepts, relations and facts. The classifier helps maintain the organization of the knowledge base, and as we will describe, we use it to match problem solving methods with goals. However, to be able to analyze the interdepende ncies between the conceptual structure of a system and its problem solving methods we need a highly declarative representation for problem solving knowledge as well. The next section describes our representation of problem sol ving knowledge.

2.3.2 Capturing Intent in Problem Solving

Problem solving knowledge in EXPECT is represented as strategies (called *methods*) for achieving particular goals. Each method has a *capability description* associated with it, which states what the method can do (e.g. "evaluate a COA"), and a *method body* that describes how to achieve the advertised capability of the method. The steps can post further subgoals for the system to achieve . The language used in the method bodies allows sequences of steps and conditional expressions.

One of EXPECT's architectural features that helps users make modifications to problem solving knowledge is a rich representation for goals and method capabilities that provides an explicit representation of intent. This represent ation makes it easier for people to understand what a method is supposed to be doing, and makes it easier for EXPECT to analyze a system's structure and hence provide guidance to a user in making modifications.

In most programming languages, function names have no real semantics associated with them. A good programmer may indicate what a function is supposed to do by giving it an appropriate name (e.g. clear-screen) but as far as the system is concerned, the name is just a string of characters. Furthermore, the relationship between the function name (what is to be done) and the functional p arameters (the things that will

10

```
capability:
  (find
          (obj (?t is (specialization-of
                       unloading-time)))
                  (of (?m is (instance-of
                       movement))))
result-type: (instance-of time-value)
method: (calculate
                  (obj ?t)
                  (of (available-vehicles ?m)))
```

Figure 3. A method to calculate the unloading time of a movement

be involved in doing it) is completely implicit. AI systems such as planners provide a more explicit representation of what is to be done. Usually the goal is represented as some sort of state expressed in some form of predicate logic. A problem with this representation that affects both acquisition and explanation is that it is removed from the way people think and talk about what they are doing, since protocols of people solving problems show that they use verb clauses to describe what they are doing rather than state descriptions (see for example, [Anzai and Simon 1979]).

To address these difficulties and decrease the distance between EXPECT'S internal representation and how people talk about what they do, EXPECT's representation is based on a vocabulary of *verbs*. In EXPECT, we represent both goals (what is to be done) a nd method capability descriptions (what a method can do) using a verb clause representation based on case grammar. Each verb clause consists of a verb and a set of slots (or "cases") that are the param eters. For example, the goal of evaluating a particular COA is represented as a verb clause where the main verb is "evaluate" and its (direct) object slot is filled by the COA instance (e.g., "coa-3") as follows:

```
(evaluate (obj (instance coa-3)))
```

The capability descriptions associated with methods are repre sented similarly, except that they may contain variables. For example, the following capability description would be associated with a method that has the capability to evaluate a course of action for force deployment:

```
(evaluate (obj (?c is (instance-of
                deployment-coa))))
```
where:
```
    deployment-coa is a coa
  that has force-movements
```

For matching, the goals and capability descriptions are translate d into corresponding Loom concepts. The goal above would match this capability description if coa-3 had movements of forces, since coa-3 would then be a kind of deployment-coa.

This approach gives us a rich representation for what is intended by a goal and what the capabilities of a method are. Unlike the state -based representation, goals and capabilities can be easily paraphrased into natural language.

Figure 2 shows a simple method from our system for evaluating transportation plans. It finds the unloading time of a movement by calculating the unloading time of the set of available vehicles of the COA movement. There are two things to notice about this

11

method. First, it is relatively straightforward to paraphrase the representations of the capability description and the method body into understandable natural language, because their structure mirrors natural language. Second, this method illustrates the use of two general kinds of parameters that can be passed in EXPECT. They distinguish between the kind of data that will be provided to the method and the kind of task to be accomplished by the method. They are indicated by `instance-of` or `specialization-of` in the capability description. These keywords indicate how the matcher should match these slots against the corresponding slots in goals.

`Instance-of` indicates that the slot is a *data parameter* and will match an instance of the indicated type. Thus, `(instance-of movement)` will match a movement instance or an instance that is more specialized. `Instance-of` slots work much like function parameters in conventional programming languages: they supply the data that the function manipulates. However, in EXPECT, slots on goals do more than just provide data; they can also further specify the task to be done. *Task par ameters* are indicated by `specialization-of` slots and match against *concepts* that appear in corresponding slots in the goal. For example, `(specialization-of unloading-time)` will match `unloading-time` or any of its specializations. The capability description of the method in Figure 2 will match a goal such as:

```
(find (obj (specialization-of unloading-time))
      (of (instance movement-23)))
```

or it will match the goal:

```
(find (obj (specialization-of
                emergency-unloading-time))
      (of (instance movement-23)))
```

Notice the "obj" slot in both cases does not *supply* data but instead it *specifies* the sort of information that is su pposed to be found by the method. `Specialization-of` slots add an additional dimension for method abstraction, allowing us to re-use the same method in several different contexts, and they are one of the ways we achieve "loose - coupling" in EXPECT, which we will discuss in detail in Section 2.4.2. In the example in Figure 2, `?t` is bound to the concept in the goal that matches `(specialization-of unloading-time)`. The variable `?t` is then used in the method body to pass the goal context on to subgoals so that the method body will compute for example the `emergency-unloading-time` rather than the `unloading-time` if emergency time was spec ified in the original goal.

The key features of our goal representation are: 1) it is structured, 2) a richer representation for intent is provided because the structure is based on a verb clause representation, and 3) in addition to data parameters, task parameters are explicitly distinguished.

2.4. Bringing Knowledge Back Together: Loose Coupling in Expect

We have argued that by separating different kinds of knowledge and providing explicit representations for them knowledge based systems created withi n the EXPECT framework can be easier to understand, and because the separation provides increased

modularity, they are easier to augment and modify. In this section, we describe how the program instantiator works to match up and integrate different knowledge sources. We refer to the matching process we use as *semantic match* because resources are matched up based on their *meaning* as opposed to their syntax. We argue that this loosens the coupling between knowledge sources, which can have distinct advantages for knowledge re-use and acquisition.

2.4.1 Resolving Goals: Semantic Matching And Goal Reformulation

In many systems, matching of goals and methods is done on a fairly syntactic basis. Lexemes in goals must match those in methods, and variables are m atched by position. In EXPECT, we have tried to move toward a matching process that is based on the semantics of the goals rather than their syntax, and one in which reformulation can be used to achieve a match when a more direct match is not possible. In EXPECT, the ability to provide looser coupling between goals and method capabilities depends on the way that they are represented. As we described in Section 2.3.2, both goals and method capabilities are represented as verb clauses. A main verb states what is to be done, using a number of slots that act as "cases" (as in case grammar). For matching, both goals and method capabilities are translated into Loom concepts that mirror their structure.

Semantic Match One of the mechanisms EXPECT provides to achieve looser coupling is based on Loom's classifier (one of the reasoners that Loom provides). Given an existing hierarchy of Loom concepts (and their definitions) organized according to the subsumption relations (A.K.O.) between them, the classifier is capable of figuring out where in the hierarchy a new concept belongs, based solely on the definitions of the concepts. To find possible methods for accomplishing a goal, the Loom classifier is used to find those methods whose capability descriptions subsume the goal. The classifier provides a form of semantic match, because match is based on the meaning of concepts, not on their syntactic form. Section 2.3.2 described how goals are represented as Loom concepts and showed some simple examples of how the semantic matcher works.

The semantic matcher finds methods whose capabilities subsume the posted goal not only when it is given the class name but also when it is given a description of the type of object that needs to be matched. An example of thi s kind of matching occurs in our network diagnosis domain. EXPECT's terminological knowledge contains the definition:

```
lanbridge-23 is a COMPONENT
                    that is CONNECTED-TO 2 networks

CONNECTED-COMPONENT is a COMPONENT
                    that is CONNECTED-TO a NETWORK
```

When EXPECT is reasoning with its problem solving knowledge about how to achieve the goal:

```
(diagnose (obj (instance lanbridge-23)))
```

13

It will be able to match that goal with a method with the capability description:

```
(diagnose    (obj    (?c    is    (instance-of    CONNECTED-
COMPONENT))))
```

because the classifier can figure out that `lanbridge-23` is a kind of component which is connected to a network, based on the definitions of `lanbridge-23` and `CONNECTED-COMPONENT` above. This kind of subsumption matching allows EXPECT to reason about the semantics of a goal in terms of the *meaning* of its parameters.

We described earlier how EXPECT relies on LOOM's classifier to automaticall organize concepts in an AKO lattice. EXPECT also relies on the LOOM classifier to reason about what goals and capabilities subsume others. This is achieved by turning goals and capabilities into LOOM descriptions. EXPECT has a core set of Loom definitions that are used for this, and include **action name** (its subclasses are essentiall verbs), **action role** (its subclasses are OBJ and any parameter name), **goal**, and **capability**. Action roles are relations whose domain is an action name, and whose range can be any existing concept in the domain (ex: ship, number) qualified by its parameter type (set or element, concept or instance, extensional or intensional). For example, the goal to compute the factorial of a number is expressed in EXPECT as:

```
(compute
     (obj (spec-of factorial)
     (of (inst-of number)))
```
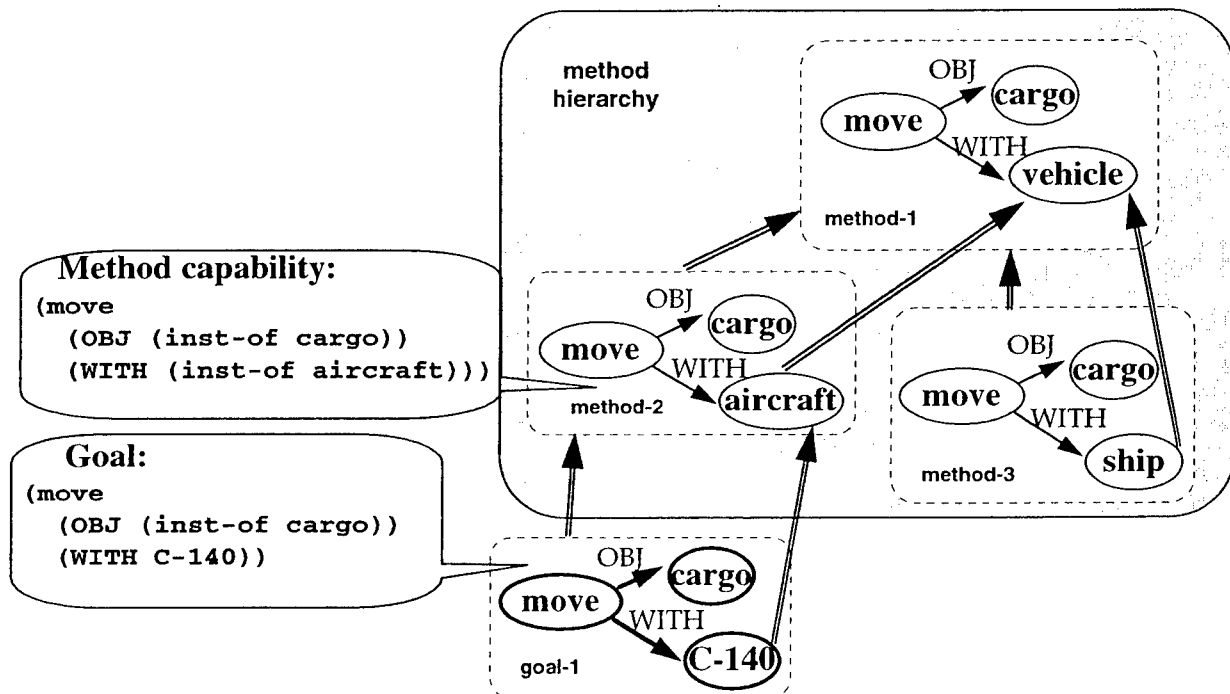

Figure 1: Translating Goals and Capabilities to Loom to organize and retrieve methods

The corresponding Loom definition that is created is:

```
(defconcept CM20
    :is (:and compute
            (:the obj (:and concept-desc
                            factorial))
            (:the of (:and instance-desc
                            number)))))
```

LOOM's classifier is now able to reason with this definition. Every term used in the parameters have their own definitions, provided in the ontologies, and LOOM will use them in reasoning about goal subsumption. Notice that these terms and their definitions can be domain independent (e.g., `violated-constraints, maximum`) or domain dependent (e.g., `location, closure-date`).

Using the techniques just described, EXPECT creates Loom definitions for the capabilities of all the methods that are defined in the knowledge base. Loom's classifier reasons about these definitions and places them in a lattice, where more general definitions subsume more specific ones. Notice that this subsumption reasoning uses the definitions of the domain terms and ontologies that are part of EXPECT's knowledge bases. As a result, the capability of a method to "move cargo with a vehicle" will subsume one to "move cargo with an aircraft", because according to the domain ontologies vehicle subsumes aircraft. This is illustrated in the method hierarchy shown in Figure 3. As a result, EXPECT's methods are *automatically organized according to their capabilities*, and their capabilities can be compared based on their place in the lattice.

EXPECT also exploits the representation of goals and capabilities for matching method capabilities with the goals that arise during problem solving. EXPECT's matcher first translates the posted goal into a Loom concept, and then invokes the Loom classifier in order to find methods whose capability descriptions subsume the posted goal. Figure 3 illustrates this matching process for the goal of moving some cargo with a C-140 (which is a particular kind of aircraft).

Once the match has been made using the Loom representation for the goal and capabilities, the original representation is used to bind parameters in the goal to corresponding variables in the capability description. This is necessary since Loom does not support variables in concepts.

Reformulations The second mechanism that EXPECT provides for looser coupling of goals and methods is reformulation. Usually if a method cannot be found to achieve a goal using semantic match, the system will attempt to reformulate the goal and then look for methods to achieve the resulting goal(s). Goal reformulation involves decomposing a goal and then assembling a new goal (or goals) by transforming pieces of the original goal based on their meaning. To be able to perform goal reformulation, one needs an explicit, decomposable representation for the goal, definitions for the terms the goal is constructed from, and domain facts to drive the reformulation process. In EXPECT, we provide two general types of reformulations, *conjunctive* and *disjunctive*. A conjunctive

reformulation involves transforming some goal into a set of goals, where *each of the goals* in the set must be performed to achieve the intent of the initial goal. Thus, a conjunctive reformulation is a form of divide-and-conquer: it splits a problem up into subpieces that together achieve the original goal. As in divide -and-conquer, the system must find a way of recombining the results of each of the subproblems back into an appropriate result for the original goal for a conjunctive reformulation to be successful. A disjunctive reformulation may also reformulate an initial goal into several goals, but at runtime, *only one of the goals* needs to be executed to achieve the intent of the original goal.

EXPECT provides three types of conjunctive reformulations: covering, individualization, and set reformulations.

A *covering* reformulation occurs when a goal can be transformed into several new goals that together "cover" the intent of the initial goal. Suppose that the following goal is posted to estimate how many support personnel are required for a COA:

```
(estimate (obj (specification-of support-personnel)
          (for (instance-of coa)))
```

Using subsumption matching, the system would try to find methods for achieving this goal. Suppose none were found, because the system had no general method for estimating the support personnel needed for a COA. Suppose, however, that the system did have methods for estimating particular types of support personnel needed for a COA. How could these methods be found? When the system failed initially to find a method, it would then try to reformulate the goal into new goals. If the domain model contained the fact:

```
support-personnel is covered by
        airport-support-personnel and seaport-support-
personnel
```

the system could reformulate the original goal into two new goals:

```
(estimate (obj (specification-of airport-support-
personnel)
          (for (instance-of coa)))
and
(estimate (obj (specification-of seaport-support-
personnel)
          (for (instance-of coa)))
```

The system would then be able to find the two methods for estimating particular types of support personnel needed for a COA. For conjunctive reformulations, part of the reformulation process involves finding a function for re-combining the results of each of the reformulations to form the result for the original goal. The appropriate function for

combining results is determined by the type of the goal. In this example, the system adds the estimates together.

This sort of reformulation process reduces the need for different parts of the knowledge base to match up exactly, which enhances the possibilities for knowledge re-use across systems. Also, because the system explicitly reasons through the reformulation process, more of the design can be captured to support knowledge acquisition. In this case, if the user added a new type of support personnel to the knowledge base, for example `law-enforcement-personnel`, then the system would use its record of this reformulation to determine that it would be necessary to perform the reformulation over again to capture the new type of support personnel. EXPECT could detect that the user needs to provide a method for estimating the law enforcement personnel needed for a COA.

Individualization reformulations are similar to covering reformulations, except that the decompose a goal over a set of objects into a set of goals over individual objects (i.e., instances). For example, given the goal of calculating the employment personnel of the force modules in a COA:

```
      (calculate (obj (specification-of employment-
personnel))
                 (of (force-modules coa-2)))
```

and the domain fact that:

```
      force-modules of coa-2 are the instances:
            `3rd-ACR 57th-IMF CVN71-ACN
```

the system could transform the original goal into three goals:

```
      (calculate (obj (specification-of employment-
personnel))
                 (of (instance 3rd-ACR)))

      (calculate (obj (specification-of employment-
personnel))
                 (of (instance 57th-IMF)))

      (calculate (obj (specification-of employment-
personnel))
                 (of (instance CVN71-ACN)))
```

where each of these goals corresponds to one of the instances force modules of coa-2.

A third kind of conjunctive reformulation is the *set* reformulation. When no method is found to achieve a goal that has a set of objects in its parameters, EXPECT tries to solve

the goal for each element of the set in turn. For example, suppose that the following go al is posted to calculate the closure date[2] for several movements:

```
(calculate (obj (specification-of closure-date))
           (of (set-of (instance-of movement))))
```

and there are no methods that operate on a set of movements. EXPECT reformulates this goal over a set into a goal over an individual movement:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of movement)))
```

The matcher will return the method for calculating the closure date of a movement. At execution time, the system will loop over each of the movements in the set and calculate one by one the closure date of the movements that are included in the set.

Finally, EXPECT provides the *input* reformulation, which is a form of *disjunctive* reformulation. It occurs when no method can be found to handle one of the inputs to a goal, but several methods can be found that together will cover the range of possible inputs that will occur at runtime. For example, given the goal:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of movement)))
```

suppose that the method library contained no method for calculating the closure date of a movement in general, but there were methods for calculating the closure date of particular types of movements and there was a domain fact that told the system that:

```
    movement is covered by airlift-movement and sealift-
movement
```

then the system could create the goals:

```
(calculate (obj (specification-of closure-date))
           (of (instance-of airlift-movement)))

(calculate (obj (specification-of closure-date))
           (of (instance-of sealift-movement)))
```

the system would write methods for each of these goals, and then create dispatching code that would select which method to use at runtime, based on the type of the type of instance of movement that was actually passed in. Note that unlike a covering reformulation, only one of the branches of a disjunctive reformulation needs to be executed.

In summary, the loose coupling that EXPECT provides through semantic match and reformulations is crucial to our approach to knowledge acquisition. First, by moving

---

[2]The closure date is the date when all the material to be shipped has arrived at the destination.

away from syntactic matching, users can add knowledge to a system without being as concerned with issues of form. This opens up the possibility for greater knowledge re - use and eases collaborative wor k on knowledge bases. The second benefit is that by having the program instantiator reason extensively about the process of matching up goals and methods, more of the design of the knowledge based system and the interdependencies between parts of the syst em can be captured (and hence, used to form expectations for knowledge acquisition). In reformulating goals, the program writer develops a rationale for how a high -level goal can be achieved in terms of lower level goals. This sort of processing is ofte n exactly the sort of reasoning that one wants to explain and use as a basis for knowledge acquisition.

2.4.2 The Program Instantiator: Capturing Interdependencies

The EXPECT program instantiator works in a refinement driven fashion. Initially, the program instantiator starts with a high level goal that specifies what the expert system is supposed to do. For example, to create an expert system for evaluating a particular COA, the system would be given the following goal:

```
(evaluate (obj (instance-of deployment-coa)))
```

This high level goal determines the scope of the knowledge based system that EXPECT will create. The goal above would create a knowledge based system that could evaluate a deployment-COA—but nothing else. On the other hand, a goal such as:

```
(evaluate (obj (instance-of coa)))
```

would create a knowledge based system that could evaluate any COA that was in EXPECT's knowledge base (assuming appropriate problem solving knowledge was also available). Thus, a single EXPECT knowledge base can be used to create a variety of knowledge based systems, each scoped to cover a different (or possibly overlapping) set of problems.

Instances that appear in goals during the program instantiation phase act as "place holders" for the actual data object that will appear when the program is executed. The use of a more general or abstract instance results in the creation of a system that can handle a wide range of inputs, i.e., all the instances of that type.

When a goal is posted, the program instantiator .sea rches its library of problem solving knowledge to find a method whose capability description matches the goal. This matching of goals and methods is a critical step in the program instantiator's reasoning and was the main topic of the previous section. H ow it is done, and the representations that are used, have a direct effect on how maintainable and reusable the knowledge base will be and EXPECT's ability to acquire new knowledge.

Once a method is selected to achieve the posted goal, the variables in the method's capability description are bound to corresponding instances and concepts in the goal. The body of the method is expanded by plugging in the bindings for the variables in the body and then posting its subgoals. During this process, if any of the slots of an instance are

accessed by the method, those accesses are recorded in the design history. This record of the interdependencies between factual knowledge (instances) and problem solving knowledge is later used to form the expectations that guide knowledge acquisition. For example, if the program instantiator notes that it uses a method that requires information about the possible-faults of a component, then when a new component is entered, the knowledge acquisition routines will know that information about the possible-faults of a component needs to be added.

A key advantage for knowledge acquisition of EXPECT's approach is that the program instantiator explores *all* the possible execution paths through the knowledge based systems it creates. It thus captures all the interdependencies between factual and problem solving knowledge, something which is not possible to do by analyzing execution traces, for example, since each analysis will only cover one execution path through the system.

Figure 4 shows a partial view of how the program instantiator expands goals. The top-level goal given to the system is to evaluate an instance of a deployment course of action. This is the goal posted in node n1. The matcher finds a method whose capability can achieve this goal, and the method's capability, the bindings, and the method's body are recorded in the node. After the bindings are substituted in the method body, the subgoal of evaluating the transportation factors of the COA would arise, and successive goal expansions would produce the goals in nodes n2 and n6.

n1
```
goal: (evaluate (obj (inst-of deployment-coa)))
achieved-by: (evaluate (obj (?c is (inst-of coa))))
bindings: ((?c (inst-of deployment-coa)))
method: (evaluate (obj transportation-factors)
                  (of ?c))
```

n2
```
goal: (estimate (obj (spec-of support-personnel))
                (for (inst-of deployment-coa)))
achieved-by: covering-reformulation obj
```

n3
```
goal: (estimate (obj (spec-of airport-support-personnel))
                (for (inst-of deployment-coa)))
achieved-by: (estimate (obj ?asp is (...
```

n4
```
goal: (estimate (obj (spec-of seaport-support-personnel))
                (for (inst-of deployment-coa)))
achieved-by: (estimate (obj ?ssp is (...
```

n5
```
goal: (calculate (obj (spec-of employment-personnel))
                 (of (inst-of deployment-coa)))
achieved-by: (calculate (obj (?p is (spec-of
                                     employment-personnel))
               (of (?c is (inst-of coa)))
bindings: ((?p (spec-of employment personnel))
           (?c (inst-of deployment-coa)))
method: (add (obj (personnel (force-modules ?c)))
```

n6
```
goal: (calculate (obj (spec-of closure-date))
                 (of (inst-of deployment-coa)))
achieved-by: (calculate (obj (?d is (spec-of closure-date)))
                        (of (?c is (inst-of coa))))
bindings: ((?d is (spec-of closure-date))
           (?c (inst-of deployment-coa)))
method: (calculate (obj ?d)
                   (of (movements ?c)))
```

DOMAIN-SPECIFIC KBS
+
DESIGN HISTORY

force-modules
coa
movements
jscp
deployment-coa

TERMINOLOGICAL KB

PROBLEM SOLVING KB
```
capability: (calculate (obj (?d is (spec-of closure-date)))
                       (of (?c is (inst-of coa))))
result-type: (inst-of time-value)
method: (calculate (obj ?d)
                   (of (movements ?c)))
```
```
capab: (calculate (obj (?d is (spec-of employment-personnel)))
                  (of (?c is (inst-of coa))))
result-type: (inst-of person-count)
method: (add (obj (personnel (force-modules ?c)))
```
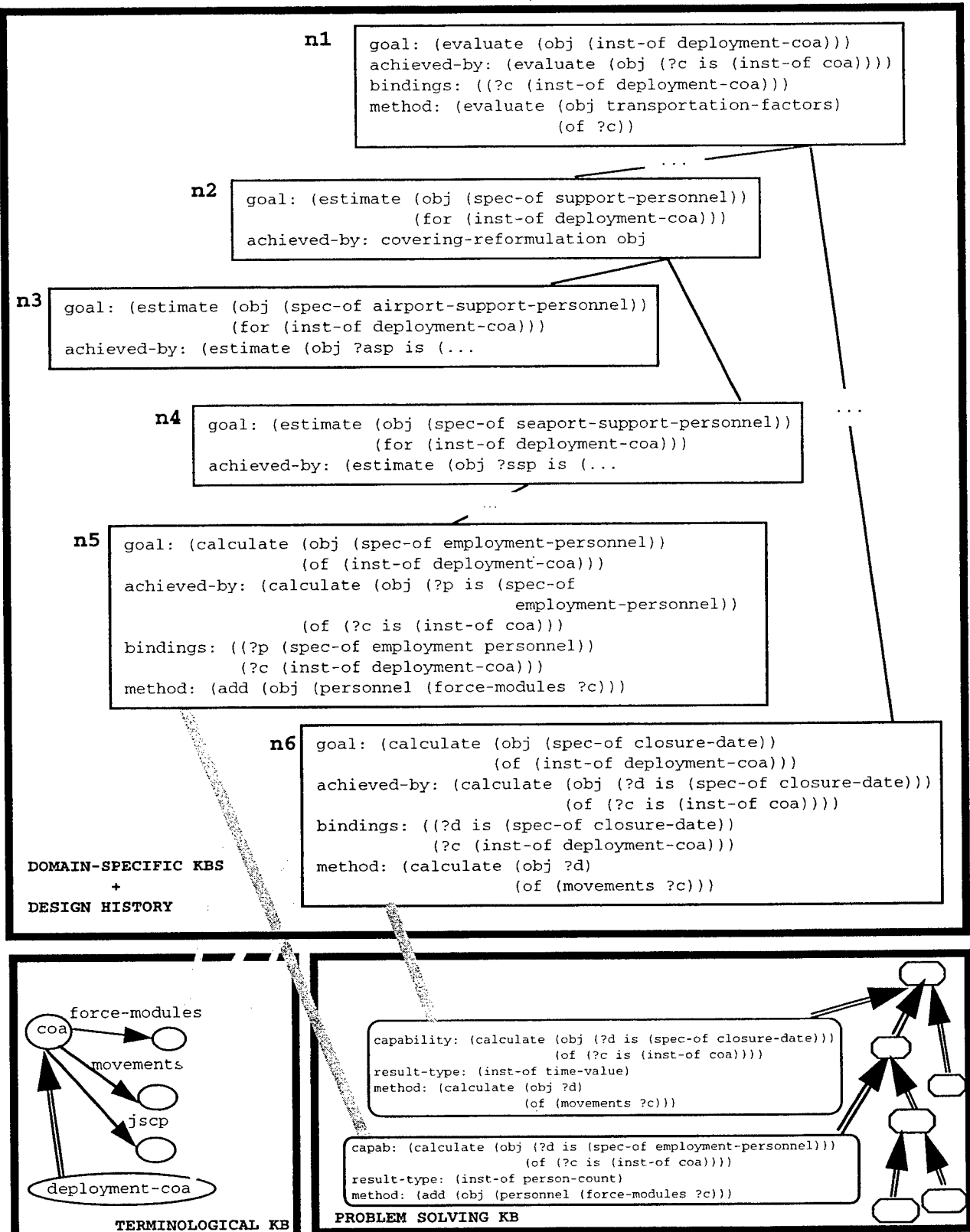
Figure 4. During problem solving, the program instantiator integrates the different types of knowledge that the diffferent knowledge bases contain and keeps track of their interdependencies (shown in the thick grey lines). These interdependencies, dynamically generated by EXPECT based on the current available knowledge, are the basis for guiding knowledge acquisition.

21

Notice that even though the method used to achieve the goal in n1 can be used for any kind of COA, the bindings specify that is used for deployment COAs and the system propagates this more specific type as it expands the subgoals. When no method is found to match a goal, EXPECT tries to reformulate the goal and try matching again. For example, the goal in node n2 is achieved by a covering reformulation of the object parameter of the goal.

During the process of expanding the goals, the program instantiator also keeps tracks of the interdependencies between the different components of the knowledge bases. Let us look more closely at node n6 in Figure 4. The goal of calculating the closure date of a deployment COA is achieved with a method. Its method body indicates that the system must calculate the closure date of all the movements of the COA. Since movements is a role of the concept COA, the system annotates that COA movements are used by the method in this node. The factual domain knowledge is effectively being linked to the problem-solving knowledge that uses it. This is shown with thick gray lines in the figure. Furthermore, the bindings indicate that the movements are used for deploy ment COAs, but not for other types of COA.

2.5. Flexible Knowledge Acquisition

By representing separately and explicitly knowledge of different types, EXPECT allows users to make changes to the knowledge bases in terms that are meaningful in the domain. By deriving the interdependencies between the different types of knowledge as they are used for problem solving, EXPECT can provide guidance for knowledge acquisition that is dynamically generated from the current content of the knowledge bases. By representing explicitly problem-solving methods, EXPECT can reason about their components and support users in modifying any component of the methods. This section describes very briefly how EXPECT's knowledge acquisition tool takes advantage of the architectural features described in this chapter, see [Gil, 1994; Gil and Paris, 1994] for a more detailed description of the knowledge acquisition tools.

Using Problem-Solving Knowledge to Guide the Acquisition of Factual Domain Knowledge

EXPECT's knowledge acquisition tool supports users in entering factual domain knowledge by automatically generating a dialogue that requires the information needed for problem-solving as indicated by the interdependencies captured by the program instantiator. Let us go back to the example in Figure 4. For example, when a user wants to define an instance of a new COA[3], EXPECT will examine the interdependencies derived by the Program Instantiator. EXPECT realizes that only deployment COAs are evaluated (according to the top-level goal in node n1), and that other kinds of COAs (such as employment COAs) are not evaluated. Based on that information, EXPECT will first request the user to be more specific about the COA. To provide maximum support to the

---

[3]In the full scale COA evaluator that we have implemented there is a wide range of factual knowledge that a user might add such as information about locations, ports, facitlities, and the characteristics of various forces.

user in providing this information, EXPECT generates a menu of options that correspond to the different kinds of COAs that are known to the system. The system continues to request the user to be more specific for as long as the subtypes of the currently specified type are used differently by the system.

Next, EXPECT examines the interdependencies to request the data needed about a deployment for problem solving. It notices that the roles used in the methods are force-modules and movements. According to the knowledge that is currently available to the system, the JSCP information is not used for COA evaluation. Thus, EXPECT requests the user to enter the force modules and movements of the COA and makes the JSCP information optional. Again, to support the user as much as possible, EXPECT generates a menu with the force modules and the movements that are currently defined in the system as suggestions for possible fillers.

If the user adds a step to a problem-solving method that uses the JSCP of a COA, EXPECT will automatically detect this new interdependency and ask the user to provide this information for any COAs.

Using Factual Domain Knowledge to Guide the Acquisition of Problem-Solving Knowledge

Suppose now that the domain knowledge changed and a new subclass of supp ort personnel was added, e.g., law-enforcement-personnel. EXPECT would then detect that to estimate the support personnel for a COA in node n2 it now generates three subgoals in the covering reformulation instead of two, and as a result it needs to have a problem-solving method for estimating the law-enforcement-personnel needed for a COA. EXPECT guides the user in specifying the new method by reusing one of the other two as a rough initial version for the new method that the user can correct by changing any of its components.

In addition to adding new problem-solving methods, EXPECT's knowledge acquisition tool supports users in modifying existing methods. For example, a user may add a new step in a method to use the priority of a COA to decide whether or not to use more resources for transportation. Suppose that the priority of a COA is not defined, and that priorities are defined as relations that only apply to mission objectives. Based on this domain knowledge, EXPECT notifies the user that the priority of a COA is not defined and thus cannot be used by a method. EXPECT also presents to the user with a menu of all the roles that are defined for COAs as options to be used in the method.

As always, the knowledge acquisition dialogue is updated if t he underlying interdependencies change. In this case, if the definition of the role "priority" is changed so that it is defined for a then the system would allow the user to use the priority of a COA.

Discussion

EXPECT's knowledge acquisition tools can:

1) derive the interdependencies between domain and problem-solving knowledge
2) use these interdependencies to guide knowledge acquisition
3) provide suggestions to the user about how to resolve a conflict based on the

nature of the interdependencies

4) justify, when asked to do so, the reasons for requesting the user's intervention based on the derivation of the interdependencies

5) support users in adding new problem solving methods and in changing the internal steps of existing ones because they are explicitly represented and their relation with domain knowledge is also specified

All these features are important for the flexibility of EXPECT's knowledge acquisition tool, and are possible because of its architectural features as they ar e described in this chapter.

2.6. Related Work

The acquisition of knowledge about a task can be viewed as a process of incorporating new knowledge into some existing knowledge structure [Rosenbloom 1988]. The existing knowledge can guide and constrain the search for new knowledge, and the process of integrating the new knowledge with the old may identify additional opportunities for learning. An acquisition system that takes this view needs to represent and understand the knowledge about the task as we ll as the process of finding and integrating new knowledge.

A general trend in research on knowledge acquisition has been to make the knowledge structures that guide acquisition increasingly explicit. In early acquisition tools many of the requirements that needed to be satisfied when adding a new piece of knowledge were not stated. The result was that they could not provide very precise guidance for acquisition. Later, more of these requirements were made explicit, but they were embedded within the acqu isition tools themselves. This allowed the tools to provide more specific guidance, but the requirements that were embedded in the tools could not be changed, which meant that some important aspects of the knowledge-based system being built could not be changed. More recent work has focused on making these requirements explicit and represented outside of the tool itself. A result of this has been more flexible acquisition tools that allow users to make a greater variety of changes to the knowledge-based system being built.

To make this more concrete, we will begin by briefly reviewing three well -known acquisition systems: TEIREISIAS [Davis 1976], SALT [Marcus 1988], PROTÉGÉ -II [Musen and Tu 1993]. We will then describe EXPECT, the acquisition framework we have been developing [Swartout and Gil 1995, Gil 1994, Gil and Paris 1994]. Each of these four systems represents a point along the trend we outlined above. We conclude with a discussion about how these systems make the knowledge structures they use f or acquisition more or less explicit, and summarize the implications of these differences in terms of the kinds of knowledge acquisition they can support.

2.6.1 Symbol-Level Approaches

TEIREISIAS [Davis 1976] was one of the earliest knowledge acquisition systems. It was designed to help a user correct and extend MYCIN's knowledge base [Buchanan and Shortliffe 1984] for diagnosing infections. If MYCIN either incorrectly concluded that a disease was present, or missed a correct diagnosis, TEIREISIAS would walk the user through the trace of rule firings to determine where the error arose. If an incorrect diagnosis was concluded, TEIREISIAS would display the rule that led to the conclusion, and ask if any of the conditions on the rule needed to be changed. If so, the user was given the opportunity to make the change. If the rule was correct, but fired because some if its conditions were incorrectly asserted, the process would recurse and the user could look at the rules that made those assertions to dete rmine if they were correct. Similarly, if a correct diagnosis was missed, TEIREISIAS would display the rule that could have caused that diagnosis to be reached, and would walk the user through the process of determining why those rules did not fire and co rrecting the problem, either by changing the conditions on the rules or by adding additional rules. When a new rule was added, TEIREISIAS provided guidance based on rule models derived from the existing rule base through statistical conceptual clustering. If rules that concluded about a particular parameter x frequently mentioned some other parameter y in their antecedents, then TEIREISIAS would point out a possible error if the user left the parameter y out in a rule concluding about x. This is useful, but a long way from actually understanding the role that the rule plays in problem solving.

TEIREISIAS understood MYCIN at the symbol level [Newell 1982]. It understood rule patterns and why a particular rule fired or did not fire, but it did not have a g lobal view of the overall algorithm that MYCIN was following. Indeed, work on TEIREISIAS pre-dated Clancey's analysis of MYCIN that showed that it was following the general problem solving strategy that he identified as heuristic classification [Clancey 1 985]. TEIREISIAS did not capture the distinctions between rules for data abstraction, heuristic match, and solution refinement that charaterize a heuristic classification system. Since TEIREISIAS did not understand the roles that the rules played in a sy stem, it could not provide much help in guiding the user concerning the content of those rules.

2.6.2. Role Limiting Approaches

The next generation of knowledge acquistion tools represents what is called a role-limiting approach. It was based on the obs ervation that the kind of problem solving method that a system uses determines the kind of domain information the system needs [McDermott 1988]. Put another way, the role that a particular kind of knowledge plays in problem solving strongly constrains how that knowledge should be expressed — what is required for the system to function. A research goal during this stage was to try to understand a number of general methods used by knowledge based systems [Chandrasekaran, 1986], such as propose-and-revise and heuristic classification, and then

to construct a knowledge acquisition tool for a particular method. Such tools could then be used to build knowledge based systems that used that particular problem solving approach. Examples of such tools include MORE [Eshelman 1988], SALT [Marcus 1988], and ROGET [Bennett 1985]. Because these tools understood how knowledge would be used in problem solving, they could provide much more guidance in helping the user formulate the knowledge he was adding to a system correctly.

SALT is a good example of a knowledge acquisition tool based on a role -limiting method. SALT was used to develop systems that use the propose-and-revise method to construct a solution to a problem. Examples included a configurer for elevators and a flow-shop scheduler. In the propose-and-revise method, a system constructs an initial approximate solution to a problem which may have a number of aspects that still remain to be determined. It then proposes a design extension to fill in missing parts of the design, and looks for possible constraints that may be violated. If a constraint violation is detected, the system has knowledge of fixes that may be used to revise the solution and correct the problem. SALT captures this general algorithm, and unde rstands that the knowledge that needs to be acquired to support this method is knowledge of ways of extending a design, the constraints that the design must satisfy, and ways of correcting constraint violations. In other words, these are the roles that new knowledge plays within the propose-and-revise strategy. Because SALT was specifically designed for building propose-and-revise systems, it includes tools that detect and correct problems that can arise in building these systems, such as cycles in the fixes and constraints.

Role-limiting approaches center knowledge base construction on filling the roles that knowledge plays in the particular problem -solving method that they are designed for. However, the problem solving method is built into the knowledge acquisition tool itself. Thus, when one selects a tool, one also determines the problem -solving method that is employed by the application. One problem with this is that many large-scale systems are not homogeneous. That is, they do not use a single pro blem-solving method throughout—some of the application can use one technique but different techniques are needed for other parts. As a result, an acquisition tool that only supports a single method has limited applicability [Musen 1992].

2.6.3. Composable Role-Limiting Methods

Rather than embodying a single problem solving method, some knowledge acquisition environments contain a library of problem solving methods of smaller size. New applications are built by composing the overall problem -solving strategy from the smaller components of the library. Examples of systems that take this approach are SBF [Klinker et al. 1991], COMET [Steels 1990], and PROTÉGÉ-II [Musen and Tu 1993].

PROTÉGÉ-II guides acquisition based on fine -grained role-limiting methods. To build an application, a knowledge engineer configures the overall problem -solving method using the components in the method library. At the same time, he or she builds a method ontology that contains the terms that are used by the method being configu red. For example, "constraint" would be one of these terms and it would be identified as one of the kinds of knowledge needed by the component that represents the revision stage of propose-and-revise. A domain ontology that contains domain -specific knowledge is represented separately. The knowledge engineer then links the knowledge roles in the

method ontology to the domain ontology, identifying how terms like "constraint" map onto domain-specific terms. Once this mapping has been done, an automatic inter face builder uses the mapping to generate a knowledge acquisition tool that allows an end user to enter domain-specific knowledge about the domain. Through this mapping the system understands just how domain-specific knowledge will actually be used, and the interface that is constructed thus requests the knowledge that will actually be needed for problem solving.

2.6.4. Flexibility through Derived Interdependencies

A remaining problem is that while acquisition systems allow a user to make changes to a system's factual knowledge, they do not allow the user to make changes to the problem solving methods that the system employs. Tools like SALT don't allow these changes because the problem solving method is implicitly encoded in the tool itself. PROTÉGÉ-II doesn't allow such changes because its methods are pre-configured and the mapping between method and domain ontologies is fixed at system design time. Since PROTÉGÉ-II's knowledge acquisition tool is derived from that mapping, it too is fixed at design time. The idea in EXPECT [Swartout and Gil 1995, Gil 1994, Gil and Paris 1994] is to derive the interdependencies between domain knowledge and problem solving methods automatically, and to be able to re -derive the dependencies as needed when changes are made to the problem solving knowledge. This approach allows a user to modify either the domain knowledge or the problem solving knowledge. We have used EXPECT to create systems in several domains. One of these domains is military transportation planning, w here EXPECT was used to construct a system to evaluate transportation plans. We will use that domain to illustrate points in this chapter.

A diagram of the architecture is shown in Figure 1. Starting at the lower left in the figure, EXPECT's knowledge bases separate out factual knowledge and problem solving knowledge. Problem solving knowledge in EXPECT consists of a mix of general, domain independent strategies as well as some that are domain specific. EXPECT's factual knowledge includes facts about the domain and terminology that describes the domain, as well as domain-independent terms that are used by problem -solving methods. In EXPECT, each problem solving method has a *capability description* that describes what the method can do. Each method also has a body, which is a step or sequence of steps for achieving the method's capability. EXPECT's method language supports conditionals, sequences of steps and embedded steps in method bodies.

The automatic method instantiator uses partial evaluation and reformulation (see [Swartout and Gil 1995]) to derive the interdependencies between the problem solving methods and domain knowledge that are needed to guide knowledge acquisition. The method instantiator starts with a high level goal that specifies a *class* of problems for which one would like to create a system. For example, one might specify that one wanted to create a system to evaluate a transportation plan from the perspective of logistics. This high -level problem description contains "generic insta nces" which are placeholders for actual data that will be used when solving specific problems. Starting with this goal, the method instantiator searches the method library for a method whose capability matches the goal. Among those that match, the most specific one is chosen, its method body is instantiated, and the generic instances replace variables in the method. If

the method body contains any subgoals, these are recursively expanded and the process continues.

During method instantiation, the system also records how domain relations and concepts are being used by the various problem solving methods. Figure 2 shows this linkage between the domain knowledge and the expansion of the problem solving methods. In the figure, the location of a seaport is used in one of the steps in the expanded method tree and berths of a seaport is used in another. Notice that there are some relations that are defined by the domain concepts but they are not used in any of the problem solving steps, such as *piers*. We expect that it will often be the case that domain ontologies will contain some relations that are useful for some problem solving methods but not for others. In particular, this would occur if we wanted to re -use a domain ontology to support a different sort of problem solving. For example, one might want to use the same domain ontology in the related (but different) problem of scheduling transportation movements. Much of the domain information needed by the two applications would be similar, but some woul d be different. EXPECT can support such re-use because its knowledge acquisition routines can focus acquisition on just the information that is actually needed to support the particular problem solving methods in use.

EXPECT also allows a user to modify problem solving methods. For example, in some transportation planning situations, one might want to take into account additional resources that might be available. For example, in figuring out the throughput of a particular location, one might want to tak e into account not only the capacity of its seaports, but also of its marinas. In EXPECT, that could be done by modifying the method that finds the seaports of a location so that it finds the marinas as well. When that modification is made, EXPECT would re-derive the dependency structure shown in Figure 2, and it would determine that information about the marinas of a location was no longer optional, but required. Accordingly, EXPECT would ask the user for information about the marinas for all the locati ons where information about marinas had not been specified, like Los Angeles.

2.6.5. Summary

Table 1 summarizes the major issues that we have pointed out in this section. As the knowledge structures we use for learning and acquisition become richer and m ore explicit, our acquisition tools can support a broader range of problem solving methods, provide more guidance to the user and help the user make a wider variety of changes to a system.

	TEIREISIAS	SALT	PROTÉGÉ-II	EXPECT
Composing the overall problem-solving strategy	no problem-solving strategy expressed, rule chaining determines strategy	manually by KA tool's designers	manually by KBS builder by selecting methods from library	automatically assembled by method instantiator
Representation of problem-solving strategy	no problem-solving strategy expressed	hard-coded in KA tool	explicit structure manually created by KBS builder	explicit structure automatically derived
Abstraction	rule templates statistically derived from existing rule base	model of problem solving (i.e., propose-and-revise), manually designed, hard-coded in KA tool	library components correspond to models of problem-solving, manually designed, hard-coded in component's KA tool	generic problem-solving methods, manually designed, and explicitly represented
When are interdependencies factual knowledge and problem-solving methods determined?	when rule statistics are gathered	when KA tool is created	when application KBS is created	whenever the application is modified
Derivation of interdependencies between domain-dependent knowledge and problem-solving knowledge	through predicates and rule models	domain-dependent knowledge corresponds to pre-determined knowledge roles of the problem-solving method	correspondences between domain ontology and method ontology are manually specified	one single ontology represents correspondences
Range of problem solving methods supported	problems solveable by backward chaining rule-based system	propose and revise	methods that can be composed from the methods in library	methods that can be expressed in EXPECT's method language
Modifications Supported	rule modification & addition	add new data (design extensions, constraints, fixes) used by propose and revise	add/modify factual information used by problem solving methods in library	add/modify factual information; add/modify problem solving methods

Table 1: Comparing knowledge acquisition tools

2.7. Summary

In real world situations, users need to be able to adapt their tools: no one has the foresight to envision all the knowledge a system might need. Our research on EXPECT addresses that problem. By separating the different kinds of knowledge that go into a knowledge based system and automatically deriving the interdependencies between them EXPECT guides a user in modifying a knowledge based system.

3. Plan Representation and Reasoning

3.1. EXPECT: REASONING ABOUT OBJECTIVES AND GOALS

An important issue in reasoning about plans, processes, and act ivities is the description of the desired goals (or objectives) as well as which actions (or tasks, or agents) have the capability to achieve them. Typically, goals are described as a flat predicate with a predicate name and several arguments and onl y limited reasoning is done about them. The EXPECT architecture, build on top of the LOOM description logic system, uses a structured representation of capabilities that has been useful in several tasks and tools, including a problem solver, a plan e ditor, a plan evaluation and critiquing tool, and an agent matchmaker. The main features of this approach are the declarative representation of *qualification parameters*(in addition to data passing parameters) for goals and capabilities, and flexible m atching techniques that go beyond exact goal match, such as goal subsumption and reformulation. A theme of the work on EXPECT has been that the representations be understandable to users, as many of the applications developed with EXPECT are plan eval uation and critiquing systems that support human planners in several military domains.

In our approach, capabilities are represented as verb clauses using a case -grammar style of formalism. Each capability consists of a verb, that specifies what is to be done, and a number of roles, or slots, which specify the parameters to be used in the action. The parameters use terms that are defined in a domain ontology. For example, the goal of estimating the closure date of a particular transportation moveme nt would be specified roughly as:

```
estimate OBJ closure-date OF transportation-movement-1
```

Here, `estimate` is the verb, and the roles are indicated in upper case. The roles are filled by concepts and instances taken from the domain ontology.

Roles can be f illed in several different ways, which allows considerable flexibility in specifying a task to be performed. A role can be filled by a specific *instance:*

```
add OBJ 3 TO 5
```

which allows us to specify particular instances that are to be used in an action. A role can be filled by a *concept:*

```
compute OBJ (spec-of factorial) of 7
```

In this case, the concept `factorial` is used to specify the kind of task that is to be performed. The data required to perform the computation are specified as parameters (in this case the number 7), while these additional task parameters allow us to express what needs to be done with that data in an explicit way and are not strictly necessary to perform the computation itself. **The fact that roles can be used both to specify the parameters or objects that will be involved in a task***and* **to further describe or specify the task itself is one of the key capabilities that our representation supports, providing us with a rich language for specifying goals.**

Roles can be a type of an instance, as in:

```
divide OBJ (inst-of number) BY 2
```

This expresses a generic goal that can be instantiated with any elements of that type.

Roles can also be filled by extensional sets as in:

```
find OBJ (spec-of maximum) OF (42 2 99)
```

31

or they can be filled by intensional sets, where the set is described by a concept:

```
find OBJ (set-of (spec-of violated-constraint))
IN (inst-of configuration)
```

Finally, it is possible to use descriptions (which are similar to the definitions of Loom concepts) in roles:

```
estimate OBJ support-personnel
IN (and location (exactly 0 seaports))
```

This is a goal to estimate the support personnel in a location with no seaports.

This approach provides a rich language for specifying behaviors. The use of a case grammar formalism makes it relativel straightforward to paraphrase the goals into natural language, helping to make them more understandable.

Capabilities are translated into LOOM definitions, following an algorithm described in. For example, `(compute (obj (spec-of factorial)) (of (5 7)))` is translated into:

```
(defconcept compute-factorial-of-numbers
        :is (:and compute
                  (:the obj (:and concept-description factorial))
                  (:the of (:and number extensional-instance-set
                                     (:filled-by instance-name 5)
                                     (:filled-by instance-name 7))))))
```

LOOM's classifier is now able to reason with this definition. Every term used in the parameters have their own definitions, provided in the ontologies, and LOOM will use them in reasoning about capability subsumption. Notice that these terms and their definitions can be domain independent (e.g., violated-constraints, maximum) or domain dependent (e.g., location, closure-date).

Given a set of capabilities expressed in this language, a set of LOOM definitions corresponding to them can be added to the knowledge base. Loom's classifier reasons about these definitions and places them in a lattice, where more general definitions subsume more specific ones. Notice that this subsumption reasoning uses the definitions of the domain terms and ontologies that are contained in the domain knowledge bases. As a result, the capability to "move cargo with a vehicle" will subsume one to "move cargo with an aircraft", because according to the domain ontologies vehicle subsumes aircraft. The capabilities are automatically organized according to their definitions, and they can be compared based on their place in the lattice.

Subsumption matching can help find suitable capabilities when presented with a query, but in some cases no subsuming capability has been added to the knowledge base. In these cases it may be possible to fulfill the request by decomposing it expressing it in different terms. This allows a more flexible matching than is possible possible if one required an exact match for goals and methods. EXPECT supports several types of reformulations:

- A *covering reformulation* is a form of divide and conquer. It transforms a goal into a set of goals that partition the original goal. If all the goals in the set are achieved, the intent of the original goal is achieved. For example, suppose a goal of estimating support personnel has been posted, but no applicable methods have been found. Suppose further that the domain ontology indicates that the concept support personnel is partitioned into unloading personnel, seaport support

32

personnel and airport support personnel. The original goal can then be reformulated into three new goals to estimate each type of personnel in the partition.

- A *set reformulation* is like a covering reformulation except that it involves a goal over a set of objects which is reformulated into a set of goals over individual objects.

- An *input reformulation* is somewhat similar to the support that some languages provide for polymorphic operators. This kind of reformulation occurs when a goal is specified with a general parameter and no single method is available at a sufficiently general level to handle the parameter. In that case, the goal can be reformulated into cases based on the subtypes of the parameter given in the ontology.

Goal reformulations allow us to state the description of method capabilities more independently from the statement the descriptions of the goals that are posted by other methods or by the user. The benefit is a more loosely coupling between methods and tasks, i.e., between what is to be accomplished and what are possible ways to accomplish it.

These structured representations of objectives have been used in three different and related contexts that require reasoning about goals: problem-solving goals, planning objectives, and agent capabilities.

EXPECT is a reasoning system that supports acquisition of problem-solving knowledge through a number of different techniques. These include maintaining a dependency model of any knowledge-based system (KBS) that is built with EXPECT, scripting tools that can guide a user through a multi-step modification to a KBS and the use of background knowledge about generic tasks. Here we focus on EXPECT's representation of tasks and subtasks within a KBS. More details about the overall reasoning and EXPECT's knowledge acquisition tools can be found in.

The problem-solving knowledge of a KBS that is built in EXPECT consists of set of methods. Each method has a capability that declares what task can be achieved by the method, a body that describes how the capability is achieved and a return type that characterizes what the method produces. The method body is written in a programming language that includes basic constructs such as a conditional test and can also include other goals. These goals may be matched by the capabilities of other methods, in which case they will be used when the method is applied, resulting in a tree structure of methods.

EXPECT method capability descriptions for methods are specified in a similar way to goals, except that variables may appear in the capability descriptions. These are bound when the capability descriptions are matched with goals. Figure 2 shows some examples of EXPECT problem-solving methods and their capabilities

33

```
(defmethod REVISE-CS-STATE
   "To revise a CS state, apply the fixes found for
    the constraints violated in the state."
   :goal (revise (obj (?s is (inst-of cs-state))))
   :result (inst-of cs-state)
   :body (apply
          (obj (find (obj (set-of (spec-of fix)))
                     (for (find (obj (set-of (spec-of
                                       violated-constraint)))
                                (in ?s)))))
          (to ?s)))

(defmethod CHECK-CAPACITY-CONSTRAINT
   "To check the Capacity Constraint of a U-Haul
    configuration, check if the capacity of the rented
    equipment is smaller than the volume to move."
   :goal (check (obj CapacityConstraint)
                (in (?c is (inst-of uhaul-configuration))))
   :result (inst-of boolean)
   :body (is-smaller
          (obj (r-capacity (r-rented-equipment ?c)))
          (than (r-volume-to-move ?c))))

(defmethod APPLY-UPGRADE-EQUIPMENT-FIX
   "To apply the Upgrade Equiment Fix in a U-Haul
    configuration, upgrade the rented equipment."
   :goal (apply (obj UpgradeEquipmentFix)
                (to (?c is (inst-of uhaul-configuration))))
   :result (inst-of uhaul-configuration)
   :body (upgrade (obj (spec-of rented-equipment-var))
                  (in ?c)))
```

Figure 2: Problem-solving knowledge in EXPECT.

Because it uses structured representations of method capabilities is, EXPECT can reason about how different methods relate to each other. This is useful for organizing method libraries as well as to support the acquisition of new problem-solving methods.
These representations support natural language paraphrasing, which is useful to develop adequate knowledge acquisition tools accesible to end users with no logic or programming background.
These goal representations have been used for almost a decade within EXPECT to develop several applications of considerable size. Two large knowledge bases were developed for the Challenge Problems of the DARPA High Performance Knowledge Bases. A knowledge based system to generate and assess enemy workarounds to target damage was developed for the 1998 Challenge Problem evaluation. This system showed the best performance and was the only one to attempt full coverage of the task. A tool for critiquing Army Courses of Action was developed together with other HPKB participants for the 1999 Challenge Problem, where each component system used shared

34

ontologies to address disjoint subsets of the critiques. Figure 3 shows the sizes of some of these knowledge bases as well the average matcher performance.
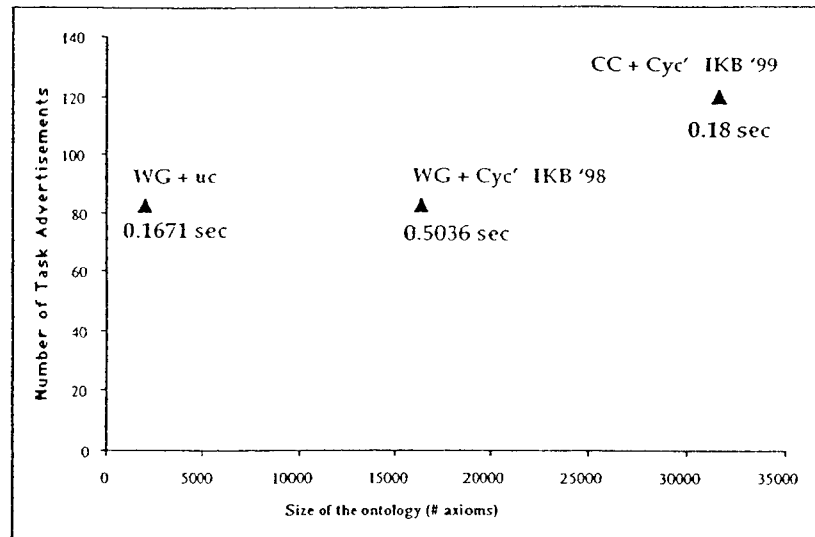

Figure 3: Performance of the Loom-based EXPECT Matcher.

A second use of these structured representations is to represent plans in the application domain that EXPECT reasons about. Plan are described as concepts, and any properties are expressed as roles and constraints. Goals are represented as Loom concepts, with each parameter corresponding to a role as described above. EXPECT was used to develop several knowledge-based systems for military air campaign planning, including offensive, defensive, and logistics aspects of air operations . Air campaign plans are composed of objectives that are decomposed into subobjectives all the way down to specific missions. Air objectives are related to one another by temporal constraints (before, after), but they lack precondition and effects information. The Air Campaign Planning Tool (ACPT) was the first of a series of plan editors that allow a user to define objectives and decompose them into subobjectives, possibly invoking automated plan generation tools to flush out the plan at the lower levels. Users entered objectives as strings with ACPT, which lack the structure that automated tools need in order to reason about them. EXPECT's goal representations provided that structured formalization of air campaign objectives, which turned out to improve the editor and to be useful to other air campaign planning tools as well. Operational users found the structured representations very useful, because an editor would enable them to be more precise in representing objectives and because they resulted in standard statements of objectives that could be shared and understood by everyone. For example, ACPT allowed them to state an objective as "conduct operations", which is too vague, or "gain air superiority", which is imprecise because it does not specify the geographical area within the theater that is intended. Several editors that make use of our grammars were built, the most widely used one is the Mastermind objectives editor, shown in Figure 4. This editor is built using Adaptive Forms, a GUI allowing a user to enter a sentence from a context-free grammar. The grammar is generated from the Loom representations of objectives.

35

Figure 4: A plan editor is used to enter objectives for an air campaign plan objectives.

The structured representation of air campaign objectives was used also for reasoning about the plan to support mixed-initiative plan creation. Because the plan creation process is mostly manual (at least at the higher levels), it is prone to error. This is aggravated by the size of the plans (several hundreds of interdependent objectives and tasks) and by the number of different people involved in its creation. In order to help users detect potential problems as early as possible in the planning process, we developed INSPECT, a knowledge-based system built with EXPECT that analyzes a manually created air campaign plan and checks for commonly occurring plan flaws, including incompleteness, problems with plan structure, and unfeasibility due to lack of resources. For example, INSPECT would point out that if one of the objectives in the plan is to gain air superiority over a certain area then there is a requirement for special facilities for storing JPTS fuel that is currently not taken into account. Without the tool, this problem might only be found when a logistics expert checks the plan, a day or more after the problem arises. INSPECT reasons about the kinds of objectives in the plan, notices that gaining air superiority requires providing reconnaissance missions, which are typically flown in aircraft that need specific kinds of fuel that need to be stored in special facilities.

A more recent application area of EXPECT's structured representations of goals is agent matchmaking. Multi-agent architectures typically offer matchmaking services that an agent can query to find what other agents can perform a given task. For example, a route planning agent may invoke threat detection agents in order to make a safe choice among

all possible routes. Typically, simple string matching suffices since the agent communities are relatively small and the agents that need to issue a request can be told beforehand what other agents are available and how they have to be invoked. In addition, most current multi-agent systems assume that an agent can perform a few tasks (often just a single task), where the advertisements and invocations of agents are negotiated in advance by the agent designers and thus can be significantly simplified. In large and heterogeneous communities of agents, where the agent that formulates the request would have no idea of whether and how another agent has advertised relevant services, there is a need for more sophisticated matchmaking mechanisms. In this kind of environment, it is likely that most requests will will be imprecise or even ill-formulated, and advertisements of agent capabilities need to be able to 1) support the requesting agent in formulating the request properly, 2) support negotiation and brokering, and 3) adapt dynamically as the agents or their environment change over time. A language is required to support descriptions of agent capabilities that enable communication among agents that had no previous knowledge of each other and thus need to provide enough information about themselves to agree to joint activities. The kinds of structured representations discussed in this chapter provide a richer language for advertising the capabilities of agents and would support more flexible matching algorithms. The application domain for this work is the integration of agent organizations and human organizations. Typical tasks in this environment involve planning a schedule for a visitor, setting up and attending meetings, and organizing off-site demonstrations and visits. Researchers, students, technical support personnel, and project assistants play different roles in each of these tasks and each person has different capabilities to offer in the organization. For example, only certain project assistants can process receipt reimbursements, only researchers can take visitors out to lunch, and onl certain people within a project are involved with relevant aspects of the software and can give demos of it. Agent capabilities are advertised using EXPECT's structured representations:

```
((capability (process (obj (spec-of reimbursement))
               (for (?r is (set-of (inst-of receipt))))))
   (agents (katya fanny tanya)))

((capability (demo (obj Phosphorus)))
   (agents (surya)))

((capability (take (obj (?v is (inst-of visitor)))
                    (to (spec-of lunch))))
   (agents (tambe knoblock minton chalupsky gil)))

((capability (setup (obj (?v is (inst-of vcr)))))
   (agents (chris ken)))
```

Using ontologies, we represent information about projects, (their members, funding agencies, software, etc.), equipment, etc. The agent capabilities are translated into Loom descriptions as we described earlier. The matchmaker uses subsumption, reverse subsumption, and several kinds of reformulations to find agents relevant to a request.

3.1.2 CLASP: CLASSIFICATION OF SCENARIOS AND PLANS

CLASP is a knowledge representation system that builds on description logic to support plan subsumption and classification . It was implemented as an extension of CLASSIC and was integrated with the LASSIE software information system . A central contribution of CLASP is the subsumption and classification algorithms for plans described as action networks. Its main application was to described the behavior of a Private Branch Exchange (PBX) switching product. The examples in this section are taken from that application as described in .

CLASP used a STRIPS-like representation of actions in the plan, and assumes a propositional representation of planning problems with conjunctive expressions of preconditions and states. Figure 1 shows the core definitions of actions, states, and plans. Actions and states are defined as CLASSIC objects, Plans are defined in CLASP's language, and described as networks of actions that achieve a goal from a given initial state. The action networks are partially ordered plans that include iteration and branching, and are called PLAN-EXPRESSION. A PLAN-EXPRESSION can be described with the constructs SEQUENCE, LOOP, REPEAT, TEST (conditional branching), OR (disjunctive branching), and SUBPLAN. The SUBPLAN construct supports modular definitions of plans through definitions of meaningful sub-networks.

```
(DEFINE-CONCEPT Action
    (PRIMITIVE
        (AND Classic-Thing
                (AT-LEAST 1 Actor)
                (ALL ACTOR Agent)
                (EXACTLY 1 PRECONDITION)
                (ALL PRECONDITION State)
                (EXACTLY 1 ADD-LIST)
                (ALL ADD-LIST State)
                (EXACTLY 1 DELETE-LIST)
               (ALL DELETE-LIST State)
               (EXACTLY 1 GOAL)
                (ALL GOAL STATE)))))

(DEFINE-CONCEPT State
    (PRIMITIVE Classic-Thing))

(DEFINE-PLAN
    Plan
    (PRIMITIVE
        (AND Clasp-Thing
                (EXACTLY 1 INITIAL)
                (ALL INITIAL State)
                (EXACTLY 1 GOAL)
                (ALL GOAL State)
                (EXACTLY 1 PLAN-EXPRESSION)
                (ALL PLAN-EXPRESSION
                        (LOOP Action)))))
```

Figure 5: CLASP definitions of actions, states, and plans.

Domain-specific types of states, actions, and plans are described using these core
definitions. Figure 5 shows some of the definitions in the telephony switching domain.
A connect dialtone action is defined as an action performed by a system (not by a user)
that provides a dialtone when the phone is off the hook and idle. Subtypes of the class
plan can be defined to create a taxonomy of plan types. For example, a plan for POTS
(Plain Old Telephone Service) can be defined as one where a caller picks up the phone
and dials, if the callee's phone is off hook the caller gets a busy signal and hangs up
otherwise the call proceeds. Notice that Originate-And-Dial-Plan is a subplan that is
defined separately and its plan expression is inserted in the appropriate node of the
POTS plan expression.

```
(DEFINE-CONCEPT System-Act
    (AND Action
         (ALL ACTOR System-Agent)))

(DEFINE-CONCEPT Connect-Dialtone-Act
    (AND System-Act
         (ALL PRECONDITION
              (AND Off-Hook-State
                   Idle-State))
         (All Add-LIST Dialtone-State)
         (ALL DELETE-LIST Idle-State
         (ALL GOAL
              (AND Off-Hook-State
                   Dialtone-State))))

(DEFINE-CONCEPT Callee-Off-Hook-State
    (PRIMITIVE State))

(DEFINE-CONCEPT Callee-On-Hook-State
    (PRIMITIVE State))

(DEFINE-CONCEPT Callee-Off-Caller-On-State
    (AND Callee-Off-Hook-State
         Caller-On-Hook-State))

(DEFINE-PLAN Pots-Plan
    (AND Plan
     (ALL PLAN-EXPRESSION
          (SEQUENCE
            (SUBPLAN
                Originate-And-Dial-Plan)
            (TEST
               (Callee-On-Hook-State
                (SUBPLAN Terminate-Plan))
               (Callee-Off-Hook-State
                (SEQUENCE
                    Non-Terminate-Act
                    Caller-On-Hook-Act
                    Disconnect Act)))))))

(DEFINE-PLAN
    Originate-And-Dial-Plan
    (AND
      Plan
     (ALL PLAN-EXPRESSION
          (SEQUENCE
              Caller-Off-Hook-Act
              Connect-Dialtone-Act
              Dial-Digits-Act))))
```

Figure 6: CLASP actions, states, and plans in the telephony domain.

Specific plans, states, and actions are created as instances of the classes defined in these taxonomies of states, actions, and plans. Specific plans are called *scenarios*, and they

40

reflect different linearized sequences of actions that can be executed in the world. Figure 6 shows a scenario where the caller picks up the phone, gets a dial tone, dials and gets a busy signal, and hangs up causing the system to disconnect. The initial and goal states are defined as instances. Specific actions are defined as instances as well, for example connect-dialtone-on-u1 is performed by a switching s stem and requires the user to be idle.

```
(CREATE SCENARIO
      pots-busy-scenario
      (AND Plan
            (FILLS INITIAL state-u1on-u2off)
            (FILLS GOAL state-u1on)
            (FILLS PLAN-EXPRESSION
                       (caller-off-hook-u1
                        connect-dialtone-on-u1
                        dial-digits-u1-to-u2
                        non-terminate-on-u2
                        caller-on-hook-u1
                        disconnect-u1)))).

(CREATE-IND state-u1on-u2off
      (AND state-U1on State-U2off))

(CREATE-IND connect-dialtone-on-u1
      (AND Connect-Dialtone-Act
            (FILLS ACTOR switching-system)
            (FILLS PRECONDITION state-u1off-idle)))
```

Figure 7: CLASP plan, state, and action instances in the telephony domain.

CLASP supported subsumption and classification of plans and scenarios by extending these functions provided in CLASSIC for concepts and instances. A plan description A subsumes a plan description B if the initial state and goal state of A subsume the initial and goal states of B, and if the plan expression of A subsumes the plan expression of B. The subsumption of plan expressions was defined by considering action networks as an extension to deterministic finite automata (DFA) where the transitions are CLASSIC subsumption checks. The plan expression EA of a plan class A subsumes the plan expression EB of a plan class B if the languages accepted by their corresponding DFAs are subsumed, i.e., DEA's language is a subset of DEB's language. A scenario is an instance of a plan class if the action network of the plan expression of the scenario is accepted by the DFA defined by the plan expression of the plan class.
The CLASP knowledge representation system supported several aspects of reasoning about plans, including organization of plan classes, retrieval of plan types and scenarios with description-based queries, and validation of scenarios based on type descriptions. In the telephony domain, it provided a useful to organize the representation of various features provided by telephony systems (such as the POTS feature), analyzing which features supported certain behavior patterns, and retrieving test scenarios for specific features and behaviors.

3.1.3. PLAN GENERATION

SUDO-PLANNER is a planner developed to reason about tradeoffs in decision making under uncertainty, and was developed in the context of medical domains. SUDO-PLANNER represented actions and plans using NIKL. Actions were represented as concepts, with their parameters as roles with the corresponding constraints represented as role restrictions. A taxonomy of action types enabled SUDO-PLANNER to exploit inheritance and classification.

TINO is a mobile robot that uses description logic to generate high-level plans. The representation of the domain includes *static axioms*, used to represent background knowledge that does not change as actions are executed, and *dynamic axioms* that represent the changes caused by the actions. Conditional plans are generated, and during execution different branches can be selected based on sensory feedback.

3.1.4. SUMMARY AND FUTURE PROSPECTS

Description logics have been applied to several areas of planning, including plan analysis, plan generation, plan recognition, and plan evaluation and critiquing. The applications range from military planning, to telephony systems, to mobile robot control. Subsumption reasoning and classification support sophisticated reasoning about general types of actions and plans than other planning research In comparison with other planning research, these systems support more sophisticated reasoning about general types of actions and plans thorugh subsumption reasoning and classification. Another advantage is that their plan representations are integrated with the description logic representations of the objects in the domain, in comparison with the more empoverished representations used in other planning research. The challenge of reasoning about plans in knowledge-intensive environments is already being raised in military planning, physical sciences research, enterprise and process modelling and management, and space operations. As the planning community continues to tackle more practical and ambitious tasks, description logics will be able to provide the kinds of knowledge representation and reasoning capabilities that these applications require.

3.2. PLANET: An Ontology for Representing Plans

As we develop larger and more complex intelligent systems in knowledge-intensive domains, it becomes impractical and even infeasible to develop knowledge bases from scratch. Recent research investigates how to develop intelligent systems by drawing from libraries of reusable components that include both ontologies and problem-solving methods. This paper introduces PLANET [4], a reusable ontology for representing plans.PLANET complements recent efforts on formalizing, organizing, and unifying A planning algorithms by focusing on the representation of plans, and adds a practical perspective in that it is designed to accomodate a diverse range of real-world plans (including manually created ones). As more complex planning systems are developed to operate in knowledge-intensive environments, ontologies present an approach to enable richer plan representations.

We have drawn from our past experience in designing, developing and integrating planning tools, and expect PLANET to ease these tasks in the future in three ways. First, we have already found it useful for *knowledge modelling*. By providing a structure that formalizes useful distinctions for reasoning about states and actions, a knowledge engineer can find the semantics of informal expressions of plans (e.g., textual or domain-specific) through designing mappings to the ontology. Reports on efforts to model plans in various application domains indicate the difficulties of representing real-world domains, and point out the need for better methodologies for knowledge modelling for planning and for richer representations of planning knowledge. We believe that PLANET takes a step in that direction. Second, a plan ontology can be a central vehicle for *knowledge reuse* across planning applications.PLANET contains general, domain-independent definitions that are common and useful across planning domains. To create a plan representation in a new domain, these general definitions can be used directly and would not need to be redefined for every new domain. Onl domain-dependent extensions will need to be added. Third, PLANET should facilitate integration of planning tools through *knowledge sharing*. Currently, practical efforts to integrate planning tools are done by designing separate interchange formats for (almost) each pair of tools, since designing a more universal format is costly and often more difficult than designing the entire set of pairwise formats. These difficulties are in part because these systems include decision-support tools such as plan editors, plan evaluation tools, and plan critiquers, which represent plans in ways that are different from traditional AI plan generation systems. An ontology like PLANET can provide a shared plan representation for systems to communicate and exhange information about the plan, and can facilitate the creation of a common, overarching knowledge base for future integrations of planning tools. An example of a successful integration of planning tools through a knowledge base is shown in.

PLANET makes the following representational commitments to provide broad coverage. First, planning contexts that refer to domain information and constraints that form the background of a planning problem are represented explicitly. Planning problems, which supplement the context with information about the initial state of the world and the goals, are represented explicitly and are accessible from the context. Alternative plans

---

[4]PLANET: a PLANsemantic NET

themselves are then accessible from each planning problem for which they are relevant. Second, PLANET maintains an explicit distinction between *external constraints*, which are imposed on a context or planning problem externally to a planning agent (including user advice and preferences), and *commitments* which the planning agent elects to add as a partial specification of a plan (for example, a step ordering commitment). The current version of PLANET does not represent aspects related to the execution of plans and actions, adversarial planning, or agent beliefs and intentions.

We present the main definitions in PLANET, including initial planning context, goals, actions and tasks, and choice points. Next, we describe three specializations of PLANET for three real-world domains where plans are of a very different nature. We conclude with a discussion of related work and some anticipated directions for future work.

This section describes how different aspects of a plan are represented in PLANET. As a convention, we use boldface to highlight terms that are defined in PLANET when they are first introduced and described in the text. Figure 1 shows a diagram of the major concepts and relations in the ontology.

Figure 1: An overview of the PLANET ontology. Arrows pointing into space represent relations whose ranges are not fixed in the ontology.

3.2.1. PLANNING PROBLEMS, SCENARIOS, AND CONTEXTS

A **planning problem context** represents the initial, given assumptions about the planning problem. It describes the background scenario in which plans are designed and must operate on. This context includes the initial state, desired goals, and the external constraints.

A **world state** is a model of the environment for which the plan is intended. When using a rich knowledge representation system, the state may be represented in a context or microtheory. A certain world state description can be chosen as the **initial state** of a

given planning problem, and all plans that are solutions of this planning problem must assume this initial state.

The **desired goals** express what is to be accomplished in the process of solving the planning problem. Sometimes the initial planning context may not directly specify the goals to be achieved, instead these are deduced from some initial information about the situation and some abstract guidance provided as constraints on the problem.

We make a distinction between *external constraints* imposed on planning problems and the *commitments* made by the plan. **External constraints** may be specified as part of the planning context to express desirable or undesirable properties or effects of potential solutions to the problem, including user advice and preferences. Examples of external constraints are that the plan accomplishes a mission in a period of seven days, that the plan does not use a certain type of resource, or that transportation is preferably done in tracked vehicles. Commitments are discussed later.

The initial requirements expressed in the planning problem context need not all be consistent and achievable (for example, initial external constraints and goals may be incompatible), rather its aim is to represent these requirements as given. A plan ma satisfy or not satisfy external constraints. PLANET represents these options with multiple **planning problem** for each planning problem context, which may add new constraints and goals, or relax or drop given ones. A planning problem is created b forming specific goals, constraints and assumptions about the initial state. Several plans can be created as alternative solutions for a given planning problem. A planning problem also includes information used to compare alternative candidate plans. Planning problems can have descendant planning problems, which impose (or relax) different constraints on the original problem or may assume variations of the initial state. Typically, AI planning systems assume one given planning problem and do not address this process, which is essential when working with real-world environments.

A planning problem may have a number of **candidate plans** which are potential solutions. A candidate plan can be **untried** (i.e., it is yet to be explored or tested), **rejected** (i.e., for some reason it has been rejected as the preferred plan) or **feasible** (i.e., tried and not rejected). One or more feasible plans ma be marked as **selected**. All of these are sub-relations of candidate plan.

3.2.2. GOALS, OBJECTIVES, CAPABILITIES, AND EFFECTS

A **goal specification** represents anything that gets accomplished by a plan, subplan or task. Both capabilities and effects of actions and tasks are subtypes of goal specification, as well as posted goals and objectives. Goals may be variabilized or instantiated. **State-based goal specifications** are goal specifications that typically represent goals that refer to some predicate used to describe the state of the world, for example 'achieve (at Jim LAX)', 'deny (at Red-Brigade South-Pass)' or 'maintain (temperature Room5 30)'.

Objective-based goal specifications are goal specifications that are typically stated as verb- or action-based expressions, such as 'transport brigade5 to Ryad'.

Goal specifications also include a **human readable description** used to provide a description of a goal to an end user. This is useful because often times users want to

view information in a format that is different from the internal format used to store it. This could be a simple string or a more complex structure.

3.2.3. ACTIONS, OPERATORS, AND TASKS

Plan task descriptions are the actions that can be taken in the world state. They include templates and their instantiations, and can be abstract or specific. A plan task description models one or more **actions** in the external world.

A **plan task** is a subclass of **plan task description** and represents an instantiation of a task as it appears in a plan. It can be a partial or full instantiation. A **plan task template** is also a subclass of **plan task description** that denotes an action or set of actions that can be performed in the world state. In some AI planners the two classes correspond to operator instances and operator schemas respectively, and in others they are called tasks and task decomposition patterns.

Plan task descriptions have a set of preconditions, a set of effects, a capability, and can be decomposed into a set of subtasks. Not all these properties need to be specified for a given task description, and typically planners represent tasks differently depending on their approach to reasoning about action. The **capability** of a task or task template describes a goal for which the task can be used. A **precondition** represents a necessar condition for the task. If the task is executed, its **effects** take place in the given world state. Tasks can be decomposed into **subtasks** that are themselves task descriptions. Hierarchical task network planners use task decomposition or operator templates (represented here as plan task templates) and instantiate them to generate a plan. Each template includes a statement of the kind of goal it can achieve (represented as a **capability**), a decomposition network into **subtasks**, each subtask is matched against the task templates down to primitive templates, represented as **primitive plan task descriptions**. Other planners compose plans as an ordered set of primitive **plan steps** (often called operators, as in STRIPS and UCPOP). Plan steps are specializations of primitive plan task descriptions that have some set of effects, as they are typically used in means-ends analysis planners.

Like goal specifications, plan task descriptions also include a **human readable description**. Some AI planners specify this information as a set of parameters of the task that are used to determine which subset of arguments will be printed when the plan is displayed.

Planning levels can be associated to task descriptions as well as to goal specifications. Some AI planners assign levels to tasks (e.g.,SIPE), others assign levels to particular predicates or goals (e.g., ABSTRIPS). Levels are also used in real-world domains, for example military plans are often described in different levels according to the command structure, echelons, or nature of the tasks.

46

3.2.4. PLANS

A **plan** represents a set of commitments to actions taken by an agent in order to achieve some specified goals. We define the following subclasses of plans: A **feasible plan** P is one for which *there exists* some plan that has a consistent superset of the commitments in P and will successfully achieve the goals. A **justified plan** is a feasible plan with a minimal set of commitments. A **consistent plan** is one whose commitments are consistent with each other, with what is known about the state and with the model of action. A **complete plan** is one that includes the tasks necessary to achieve the goals to the required level of detail (this depends on the planning agent's concerns). These definitions are useful to describe properties of plans and to accommodate different approaches that planners use. For example, we can represent that a hierarchical planner generates a feasible plan at each planning level, and a complete plan only at the lowest level. A partial-order planner, such as UCPOP, successively refines feasible plans until finding a solution which is a complete plan. There is no requirement for a plan to be justified or consistent in order for it to be represented in PLANET. This allows us to represent not only machine-generated plans but also human-generated plans, which are likely to contain errors.

It can be useful to state that a plan forms a **sub-plan** of another one. For example, military plans often include subplans that represent the movement of assets to the area of operations (i.e., logistics tasks), and subplans that group the operations themselves (i.e., force application tasks).

3.2.5. CHOICE POINTS, ALTERNATIVES, DECISIONS, AND COMMITMENTS

In searching or designing a plan, a number of choices typically need to be made. At a given choice point, several alternatives may be considered, and one (or more) chosen as selected. Such choices are represented in PLANET as a type of **commitment**. Commitments can be made in both plans and tasks. **Plan commitments** are commitments on the plan as a whole, and may be in the form of actions at variousl detailed levels of specification, orderings among actions and other requirements on a plan such as a cost profile. The tasks that will form part of the plan are represented as a subset of the commitments made by the plan. **Task commitments** are commitments that affect individual tasks or pairs of tasks. An **ordering commitment** is a relation between tasks such as (before A B). A **temporal commitment** is a commitment on a task with respect to time, such as (before ?task ?time-stamp). Another kind of commitment is the selection of a plan task description because it **accomplishes** a goal specification. This relation records the intent of the planning agent for the task, and is used in PLANET to represent causal links.

3.2.6. DISCUSSION

PLANET does not include representations for some entities that are typically associated with planning domains, *e.g.* agents, resources, time, and location. Different systems that reason about plans use different approaches to represent and reason about these entities. Separate ontologies for them can be developed and integrated with PLANET. We use PLANET in combination with an ontology of Allen's time relations and the OZONE resource ontology, and in combination with an ontology of plan evaluations and critiques that we have developed. For systems and domains where there is no need for complex representations of agents, resources, time, and location, it is trivial to extend PLANET with simple representations of them and we have done so ourselves for some of the domains described below.

3.2.7 Using PLANET for Real-World Domains

This section describes how we used PLANET to represent plans in three different domains. Although all three are military domains, the plans are of a radically different nature in each case. In the first two domains, plans were built manually by users and needed to be represented as given, containing potential flaws and often serious errors. In the JFACC domain, plans are hierarchically decomposed and have verb-based objectives. Information about causal links and task decomposition templates is not provided. In the COA domain, plans have a hierarchical flavor that is not alway explicitly represented in the plan. In the Workarounds domain, plans were generated automatically by an AI planner. This section describes the domains in more detail.

3.2.7.1 PLANET-JFACC

This is a domain of air campaign planning where users follow the strategies-to-tasks methodology. In this approach, users start with high-level objectives and decompose them into subobjectives all the way down to the missions to be flown. Using a plan editing tool, a user defines objectives, decomposes them into subobjectives, and can specify temporal orderings among plan steps. Some subobjectives may be handed to an automated planner to be fleshed out to lower levels. The rest of this discussion will focus on the representation of these manually created plans.

Figure 2 shows an excerpt of an air campaign plan as it would be specified by a domain expert, indicating the hierarchical decomposition through indentation. Options (marked with stars) indicate disjunctive branches of the plan that are explored as alternatives. The bottom of the figure shows how a user can specify an objective.

beginminipage3.5inO1: Eliminate enemy SSM threat to US allies by D+5
*Option1: Destroy all known enemy SSM launchers and launch facilities by D+5
O111: Destroy fixed enemy SSM launch sites by D+5

```
O1111: Destroy [them] on NW area using precision weapons
  *Option1: Destroy [them] using stealth aircraft
    O111111: Destroy [them] using F117 with GBU-27
  *Option2: Destroy [them] using SEAD aircraft
O112: Destroy storage facilities for SSM equipment by D+5
*Option2: Disrupt and disable the enemy C2 infrastructure for SSM
  O2: Airlift wounded and civilian non-combatants by D+2
Objective ID: O-152   Level: AO   Phase: II   Parents: O-98, O-61
  Statement: Maintain air superiority over NW sector
  Sequence restrictions: Before O-138, Before O-124
endminipage
```

Figure 2: An excerpt of an air campaign plan and the specification of an objective.

We represent **Air campaign plans** as a subclass of the class plan. These manuall
created plans do not capture well the rationale behind the objective decompositions. For
example, users do not indicate causal links, i.e., which effects enable the conditions
needed by other tasks. Nor do they indicate the intended plan tasks and instead capture
only the goals that these tasks are supposed to accomplish.

Air campaign objectives are verb-based statements, so we represent them as a subclass
of objective-based goal specifications. Some of their clauses are turned into constraints
on the goal, including temporal constraints (*within 21 days*), geographical constraints (*in
Western Region*), and resource constraints (*using B-52s from airbase XYZ*). Each
objective may have several children and several parents (unlike plans generated b
hierarchical AI planners where there is only one parent).**Options** indicate alternative
ways to decompose an objective, and are represented as a specialization of alternative
plans.

The decomposition hierarchy is divided into levels, including low-level air tasks and
other higher-level air objectives. Objectives belong to one **phase** of the campaign (e.g.,
deployment phase, operations phase, redeployment phase, etc.), which we represent b
grouping objectives into subplans. Each objective also belongs to an **air campaign
objective level** in the decomposition hierarchy (e.g., air tasks are considered to be at a
higher level than air activities), which we define as subclasses of planning level.

In this domain, human planners create air campaign plans in the context of an overall
military campaign and the specific guidance provided by the Joint Forces Air
Component Commander (JFACC). We define **JFACC-context** as a subclass of planning
problem context, and attach to it information such as the available resources specified in
the Air Order of Battle, the capabilities of the airbases in the theater of operations, and
the commander's guidance. This guidance includes the top level objectives of the plan as
well as rules of engagement, e.g., not to fly over certain areas, which become constraints
of the planning problem.

3.2.7.2. PLANET-COA

This is a Course of Action (COA) analysis problem in a military domain of relevance to
the DARPA High Performance Knowledge Bases Program. We developed a critiquing
tool that finds flaws in manually developed COAs for Army operations at the division
level. A COA is specified by a user as a set of textual statements (*who* does *what*, *when*,
where, and *why*), together with a sketch drawn over a map. The PLANET-COA ontology

allows us to represent the COA that results from joining both text and sketch, which is the input to our critiquing tool. An example of part of a COA textual statement follows:

On H hour D day, a mechanized division attacks to seize OBJ SLAM to protect the northern flank of the corps main effort. A mechanized brigade attacks in the north, as an economy of force, to fix enemy forces in zone denying them the ability to interfere with the main effort's attack in the south. A tank heavy brigade, the main effort, passes through the southern mechanized brigade and attacks to seize the terrain vicinity of OBJ SLAM denying the enemy access to the terrain southwest of RIVER TOWN. [...]

A typical COA includes the overall mission and a set of tasks that need to be performed divided into five categories: close, reserve, security, deep and rear (not shown here). The close statements always contains a main effort for the COA and a set of supporting efforts. In our representation, the mission defines two important features of the plan: its top-level goal (e.g., *protect the northern flank of the corps main effort*), and an indication of the top-level task to be used to accomplish that goal (e.g., *attack to seize OBJ SLAM*). We define **COA problem** as a subclass of planning-problem, make its problem goal the top-level goal indicated in the mission statement, and add the rest of the mission statement as a constraint on how to select tasks in the plan. The five task categories are represented as sub-plans (they are not subtasks or subgoals but useful categories to group the unit's activities). Each sentence in the statement is turned into a plan task as follows. There is a specification of *who* is doing the task, e.g., *a mechanized brigade*, which is represented as the agent of the plan-task. There is an indication of *what* is to be done, e.g., *attacks to fix enemy forces*, which is interpreted as a fix plan-task (where fix is a kind of task that is a subclass of the class attack). The *why* (or *purpose*) e.g., *to deny enemy forces the ability to interfere with the COA's main effort* can be a state-based ("enable P", "prevent P") or action-based ("protect another unit from enemy"). Therefore, the ontology defines the **purpose** of a COA task as a goal specification that can be either an effect or a capability of the plan-task. The *where*, e.g., *in the North* is the location of the plan task. The *when* clause (e.g., *H hour D day*) is represented as a temporal commitment or as an ordering commitment if it is specified with respect to another task. Finally, the **main effort** and **supporting efforts** are defined as specializations of the subtask relation.

The PLANET ontology also represents the context, assumptions, and situation in which the plan is supposed to work in this domain. A COA is supposed to accomplish the mission and other guidance provided by the commander, and to work in the context of the given situation as analyzed by the commander's staff, which includes terrain information and enemy characteristics. We define **COA problem context** as a subclass of planning-problem-context, and define its scenario to be composed of **commander products** and **staff products**. All COA problems are attached to this problem context.

3.2.7.3. PLANET-WORKAROUNDS

We developed a tool to aid in military target analysis by analyzing how an enemy force may react to damage to a geographic feature (e.g., a bridge or a tunnel) The possible workarounds include using alternative routes, repairing the damage, or breaching using engineering techniques such as installing a temporary bridge. Because the purpose of damaging the target is typically to delay the movement of some enemy unit or supply, it is important to estimate how long the implementation of the workaround will take. Note that this depends on what actions can be performed in parallel. The system was also

designed to show not one possible workaround plan but several options that the enem may take. An example workaround is shown in Figure 2

```
beginminipage3inOPTION B: Use Medium Girder Bridge (MGB) bridge
                    Minimum delay to enemy: 10 hrs
                    Transportation time: 4.5 hrs
                    Engineering time: 3.5 hrs
                    Required Assets:
        - MGB of Engineering Company 201 (double story, 11 bays)
                - Bulldozer of Engineering Company 201
        Substep: Move MGB to bridge site (Time: 4.5 hours)
                    - Ordering: Before assemble MGB
            Substep: Asse ble MGB (Time: 1.8 hours)
                    - Technique: Double story, 11 bays
                    - Ordering: Before emplace MGB
            Substep: Emplace MGB (Time: 0.5 hours)
                    - Ordering: Before Move unit across MGB
        Substep: Move unit across MGB (Time: 2 hours)
                            ...
                        endminipage
```

Figure 8: An example portion of a workaround plan.

We divided the problem in two. First, we used the AI planner Prodigy to generate a workarounds plan. We added information to the operators about the resources used for each step, and which resources are non-shareable. The planner then generated a partial order of workaround steps, in which unordered steps can be completed in parallel. Second we built a plan evaluation system to estimate the time that each step takes to complete and calculate the overall duration based on the partial order. This is a knowledg -based system that used several ontologies of engineering assets, units, and workaround steps and plans.
PLANET did not exist when this workarounds plan ontology was first developed, so we describe a reimplementation using PLANET. Actions are represented as primitive plan steps. The ordering commitments and resources used are straightforward to represent in PLANET. In the planner we subdivided the step parameters into those whose values affected plan correctness and those that were only used to determine the length of the plan after it was created. This distinction had not been captured in the original system.

3.2.8 Benefits of PLANET

There is not yet an agreed methodolog to evaluate the quality of ontologies and their actual use in systems (they can be present in a KB but that does not tell us how much they are actually used). They are often accepted to be useful purely on the basis of their existence. We claim that PLANET can be (and already has been) useful in knowledge reuse, modelling and sharing. This section presents some estimates that show the reuse of PLANET in the three domains discussed in this paper and describes the benefits of using it as a modelling tool in the COA domain [5].

3.2.8.1 COVERAGE AND KNOWLEDGE REUSE

We wanted to measure the amount of reuse of the general PLANET ontology in each specific domain. Here we present estimates of reuse in creating new terms, since we are

---

[5]PLANET did not exist when we worked on the JFACC and workarounds domains.

interested in understanding the generality and coverage of PLANET. To do this, we estimated how many axioms of PLANET were actually used in each domain, and how many new axioms were needed.

It is important to factor out domain definitions that are part of what is often described as populating the knowledge base, or knowledge base *stuffing*. For example, there may be fifty or five hundred possible tasks in a domain that share the same basic structure but this should not distort a measure of how reusable a general-purpose ontology is. For this evaluation we take these definitions out of the domain-specific ontologies and leave only those definitions that specified the basic structure of the plans. We estimated the size of each ontology by counting its axioms. We considered an axiom to be an statement about the world, including **is a** predicates, class constraints, and role constraints. We make strong use of inheritance among classes, so axioms are only stated and thus counted once.

We counted the concepts in each domain that were subconcepts of a concept in the PLANET ontology, to measure of the coverage of the domain that the ontolog provided. We estimated how many axioms of thePLANET ontology were actually used in each domain by computing the "upward closure" of the definitions in the domain ontologies. The results are as shown in Table 1 Coverage is high: on average, 82% of the concepts defined in each domain are subconcepts of concepts in PLANET. However, the proportion of the ontology used by each domain is much lower, averaging 31% of the axioms. This is not surprising. First, PLANET covers a range of planning styles, including actions with preconditions and effects and decomposition patterns, but none of the domains has all of these. Second, PLANETcan represent incremental decisions of planners, including commitments and untried alternatives, but the domains onl represented complete plans. In general we do not expect a single domain to use a high proportion of the ontology.

Domain	Axioms	Concepts	Rels	Covered concept	Coverage
Planet	305	26	37		
COA	267	58	37	39	67%
COA u.c.	106	7	12		35%
JFACC	102	15	12	12	80%
JFACC u.c.	86	9	6		28%
WA	100	13	10	13	100%
WA u.c.	91	12	4		30%

Table 1: Estimates of reuse of the PLANET ontology.

There are various other ways to reuse knowledge. One can measure the reuse of ontologies by estimating how many terms are used during problem solving or reasoning. An informal analysis of the JFACC and Workarounds domains (the problem solvers for the COA domain were under development) showed that most (if not all) the new definitions would be used during problem solving, but this should be determined empirically. The ontology is also reused in modelling a new domain. Even if a term is not used in the new system, it may still have been used to understand how to model certain aspects of the domain as we discuss next.

3.2.8.2 KNOWLEDGE MODELLING

We had developed an initial model of the COA domain without usingPLANET, and it changed significantly once we created the COAdefinitions using the PLANET definitions. Here are some examples.PLANET was useful in modelling the close/security/etc. categories. At first, they appeared to be subtasks of the COA, but we realized that the individual statements are the real subtasks in this domain, and these categories are meaningful groupings of the statements that are helpful to human planners. As a result we represented them as subplans.PLANET gave similar support for representing the main and supporting efforts which turn out to be subtasks instead of subplans. It also helped us understand how to interpret the mission statement, and its relationship with the COA main effort. The mission statement specifies top-level goals and constraints for the top-level task as we explain above.

3.3 Related Work

In creating PLANET, we have drawn from previous work on languages to represent plans and planning knowledge. These languages are often constrained by the reasoning capabilities that can be provided in practice by AI planning systems. Since PLANETis an ontology, it does not make specific commitments about the language in which various items are expressed. The planning knowledge represented in these languages can be mapped into PLANET. PLANET also accommodates plans that were not created by A planning systems, and provides a representation for the context of the planning problems that are given to these systems.
SPAR is an ongoing effort to create a standard vocabulary to describe plans that is compatible with other standards, such as the Process Interchange Format (PIF). SPAR currently comprises a set of textual statements that will be used to develop the model. CPR was developed as an overarching plan representation across various militar domains. These efforts are aimed at plan representations of a more general nature, and cover aspects of plan execution. However, as a result of their generality they would require many more extensions than PLANET to represent the domains discussed in this paper.

Related work on problem-solving methods for planning analyzes AI planning algorithms (or planning methods) and identifies the typical knowledge roles that characterize the main types of domain knowledge used by these planning methods. The main knowledge roles in this study map directly to classes in PLANET. PLANET adds many more relations between the roles and contains many more classes and axioms. PLANET was also designed from the perspective of planning environments where plans are manually created (instead of representing only plans of AI planning systems), and as a result can also represent the errors and flaws that these plans often contain. It would be useful to add to PLANET the static vs dynamic distinctions contributed by this study.

4. Deriving Expectations to Guide Knowledge Base Creation

Successful approaches to developing knowledge acquisition tools use expectations of what the user has to add or may want to add, based on how new knowledge fits within a knowledge base that already exists. When a knowledge base is first created or undergoes significant extensions and changes, these tools cannot provide much support. This chapter presents an approach to creating expectations when a new knowledge base is built, and describes a knowledge acquisition tool that we implemented using this approach that supports users in creating problem-solving knowledge. As the knowledge base grows, the knowledge acquisition tool derives more frequent and more reliable expectations that result from enforcing constraints in the knowledge representation system, looking for missing pieces of knowledge in the knowledge base, and working out incrementally the inter-dependencies among the different components of the knowled e base. Our preliminary evaluations show a thirty percent time savings during knowledge acquisition. Moreover, by providing tools to support the initial phases of knowledge base development, many mistakes are detected early on and even avoided altogether. We believe that our approach contributes to improving the quality of the knowledge acquisition process and of the resulting knowledge-based systems as well.

4.1. INTRODUCTION

Knowledge acquisition (KA) is recognized as an important research are a for making knowledge-based AI succeed in practice. An approach that has been ver effective to develop tools that acquire knowledge from users is to use *expectations* of what users have to add or may want to add next. Most of these expectations are derived from the *inter-dependencies* among the components in a knowledge-base system (KBS) and Protégé-II use dependencies between factual knowledge and problem-solving methods to find related pieces of knowledge in their KBS and create expectations fro m them. To give an example of these expectations, suppose that the user is building a KBS for a configuration task that finds constraint violations and then applies fixes to them. When the user defines a new constraint, the KA tool has the expectation that the user should specify possible fixes for cases where the constraint is violated, and helps the user do so. These tools can successfully build expectations because there is already a body of knowledge where the new knowledge added by the user must fit in. In the configuration example, there would be problem solving knowledge about how to solve configuration tasks (how to describe a configuration, what is a constraint, what is the relation between a constraint and a fix, how to apply a fix, e tc.) However, when a new knowledge base (KB) is created (or when an existing one is significantly extended) there is little or no pre-existing knowledge in the system to draw from. How can a KA tool support the user in creating a large body of new knowledge? Are there any sources of expectations that the KA tool can exploit

This chapter describes our approach to developing KA tools that derive expectations from the KB in order to guide users during KB creation. Through an analysis of the KB creation task, we were able to detect several sources for such expectations. The expectations result from enforcing constraints in the knowledge representation system, looking for missing pieces of knowledge in the KB, and working out incrementally the inter-dependencies among the different components of the KB. As the user defines new *KB elements*(i.e., new concepts, new relations, new problem-solving knowledge), the KA tool can form increasingly more frequent and more reliable expectations. We implemented a KA tool called EMeD that uses these sources of expectations to support users in adding problem -solving knowledge. Our preliminary evaluation shows an *average time savings of 30%* to enter the new knowledge. We believe it will be even higher for users who are not experienced knowledge engineers.

The chapter begins by describing why KB creation is hard. Then we present our approach, and describe the KA tool that we implemented. Finally, we show the results from our experiments with several subjects, and discuss our conclusions and directions for future work.

4.2. Creating Knowledge Bases

There are several reasons why creating a knowledge base is hard:

- **Developers have to design and create a large number of KB elements.** KB developers have to turn models and abstractions about a task domain into individual KB elements. When they are creating an individual KB element, it is hard to remember the details of all the definitions that have already been created. It is also hard to anticipate all the details of the definitions that remain to be worked out and implemented. As a result, many of these KB elements are not completely flawless from the beginning, and they tend to generate lots of errors that have unforeseen side effects. Also, until a KB element is debugged and freed from these errors, the expectations created from it may not be ver reliable.

- **There are many missing pieces of knowledge at a given time.** Even if the developers understand the domain very well, it is hard to picture how all the knowledge should be expressed correctly. As some part of the knowledge is represented, there will be many missing pieces that should be completed. It is hard for KB developers to keep track of what pieces are still missing, and to take them into account as they are creating new elements.

- **It is hard to predict what pieces of knowledge are related and how.** Since there is not a working system yet, many of the relationships between the individual pieces are in the mind of the KB developer and have not been captured or correctly expressed in the KB.

- **There can be many inconsistencies among related KB elements that are newly defined.** It is hard for KB developers to detect all the possible conflicts among the definitions that they create. Often times they are detected through the painful process of trying to run the system and watching it not work at all. The debugging is done through an iterative process of running the system, failing, staring at various traces to see what is happening, and finally finding the cause for the problem.

As intelligent systems operate in real-world, large-scale knowledge intensive domains, these problems are compounded. As new technology enables the creation of knowledge bases with thousands and millions of axioms, KB developers will be faced an increasingly more unmanageable and perhaps impossible task. Consider an example from our experience with a Workarounds domain selected by DARPA as one of the challenge problems of the High-Performance Knowledge Bases program that investigates the development of large-scale knowledge based systems. The task is to estimate the delay caused to enemy forces when an obstacle is targeted by reasoning about how they could bypass, breach or improve the obstacle. After several large ontologies of terms relevant to battlespace reasoning were built (military units, engineering assets, transport vehicles, etc.), we faced the task of creating the problem solving knowledge base that used all those terms and facts to actually estimate the workaround time. We built eighty-four problem-solving methods from scratch on top of several thousand defined concepts, and it took two intense months to put together all the pieces. Figure 1 shows some examples of our methods. Each method has a *capability* that describes what goals it can achieve, a *method body* that specifies the procedure to achieve that capability (including invoking subgoals to be resolved with other methods, retrieving values of roles, and combining results through control constructs such as conditional expressions), and a *result type* that specifies the kind of result that the bod is expected to return (more details of their syntax are discussed below). Creating each method so that it would use appropriate terms from ontologies was our first challenge . Once created, it was hard to understand how the methods were related to each other, especially when these interdependencies result from the definitions in the ontologies. Despite the modular, hierarchical design of our system, small errors and local inconsistencies tend to blend together to produce inexplicable results making it very hard to find and to fix the source of the problems. Although some portions of the knowledge base could be examined locally by testing subproblems, we often found ourselves working all the way back to our own documentation and notes to understand what was happening in the system.

(define-method M1
(documentation "In order to estimate the time that it takes to narrow a gap with a bulldozer, combine the total dirt volume to be moved and the standard earthmoving rate.")

(capability (estmate (obj)?t is (spec-of time))))
            (for (?s is (inst-of narrow-gap)))))
(result-type (inst-of number))
(body (divide (obj (find (obj (spec-of-dirt-volume))
                (for ?s))))
        (by (find (obj (spec-of standard-bulldozing-rate))
                (for ?s)))))))

(define-method M2
(documentation "the amount of dirt that needs to be moved in any workaround step that involves moving dirt (such as narrowing a gap with a bulldozer) is the value of the role earth-volume for that step.")

(capability (find (obj (?v is (spec-of dirt-volume))))
                (for(?s is (inst-of bulldoze-region))))))
(result-type (inst-of number))
(body (earth-volume ?s)))

(define-method M3
(documentation "The standard bulldozing rate for a workaround step that involves earthmoving is the combined bulldozing rate of the dozers specified as dozer-of of the step.")

(capability (find (obj (?r is (spec-of standard-bulldozing-rate)))
                (for (?s is (inst-of bulldoze-region))))))
(result-type (inst-of number))
(body (find (obj (spec-of standard-bulldozing-rate))
                (of (dozer-of ?s))))

Figure 1: Methods in a simplified workaround generation domain.

In summary, it is hard for KB developers to keep in mind all the definitions that the create and to work out their interdependencies correctly. KB developers generate and resolve many errors while they build a large body of knowledge. Our goal is to develop KA tools that help users resolve these errors and, more importantly, help them avoid making the errors in the first place.

4.3. Approach

We identified several sources of expectations that KA tools can exploit in order to guide users in creating a new knowledge base. We explain our approach in terms of the problems and examples described in the previous section.

- *Difficulty in designing and creating many KB elements RightarrowGuide the users to avoid errors and look up related KB elements.*

57

First, each time a KB element is created by a user, we can check the dependencies within the element and find any potential errors based on the given representation language. For example, when undefined variables are used in method body, this will create an expectation that the user needs to define them in the method.

In our example, Method M1 has two variables, ?t and ?s, defined in its capability, and if the method body uses a different variable, the system can send a warning message to the user. Likewise, if a concept definition says that a role can have at most one value but also at least two values, then this local inconsistency can be brought up. By isolating these local errors and filtering them out earlier in the KB development process, we can preven t them from propagating to other elements in the system.

However small the current KB is, if there are KB elements that *could be* similar to the one being built, then they can be looked up to develop expectations on the form of new KB element. For example, developers may want to find existing KB elements that are related with particular terms or concepts based on the underlying ontology. If there is a concept hierarchy, it will be possible to retrieve KB elements that refer to superconcepts, subconcepts, or given concepts and let the user develop expectations on the current KB element based on related KB elements. For example, if a developer wants to find all the methods related to moving earth, the system can find the above methods, because narrow -gap and bulldoze-region are subtypes of move-earth. When the user adds a new method about moving earth to fill a crater, then it may be useful to take them into account. Specifically, M1 can generate expectations on how a method for estimating time to fill a crater should be built.

- *Many pieces of knowledge are missing at a given time: RightarrowCompute surface relationships among KB elements to find incomplete pieces and create expectations from them*

The KA tool can predict relationships among the methods based on what the capability of a method can achieve and the subgoals in the bodies of other methods. For example, given the three methods in Figure 1, method M1 can use M2 and M3 for its two subgoals --- find dirt volume and find bulldozing rate. These relationships can create method-submethod trees that are useful to predict how methods will be used in problem solving. In the process of building this kind of structure, the system can expose missing pieces in putting the methods together. For example, *unmatched subgoals* can be listed by collecting all the subgoals in a method that cannot be achieved by any of the already defined methods. The user will be reminded to define the missing methods and shown the subgoals that they are supposed to match. In Figure 1, if a method for the subgoal of method M3 to find the standard bulldozing rate of given dozer is not defined yet, the user is asked to define one and may create one that only works for military dozer or any dozer in general.

Similarly, if a concept is used in a KB e lement definition but not defined yet, then the system will detect the *undefined concept*. Instead of simply rejecting

such definition, if the developer still wants to use the term, the KA tool can collect undefined concepts and create an expectation that the developer (or other KB developers) will define the term later.

- *Difficulty in predicting what pieces of knowledge are related and how Rightarrow Use surface relationships to find unused KB elements and propose potential uses of the elements*
  The above surface relationships among KB elements, such as method -submethod relationship can also help detect unused KB elements. If a method is not used b any other methods, then it can be collected into an *unused method list*. In addition to finding such unused methods, the KA tool can propose potential uses of it. For exam ple, if the capability of a method is similar to one of the unmatched subgoals (e.g., same goal name and similar parameter types), then a potential user of the method will be the method that has the unmatched goal.
  In the same way, concepts created but not referred to in any other definitions can be collected into an *unused concept list*. The KA tool can develop expectations of KB elements that will use the definitions or perhaps even deleting these concepts if they end up being unnecessary.

- *Inconsistencies among newly defined KB elements Rightarrow Help users find them early and propose fixes*
  The KA tool can check if the user-defined result type of a method is inconsistent with what the method body returns based on the results of the subgoals. If there are inconsistent definitions, the system will develop an expectation that user has to modify either the current method or the methods that achieve the subgoals.
  Also, for concept definitions, there can be cases where a user wants to retrieve a role value of a concept, but the role is not defined for the concept. In addition to simply detecting such a problem, the system may propose to define the role for the concept or to change the method to refer to a different but related concept that does have that role.
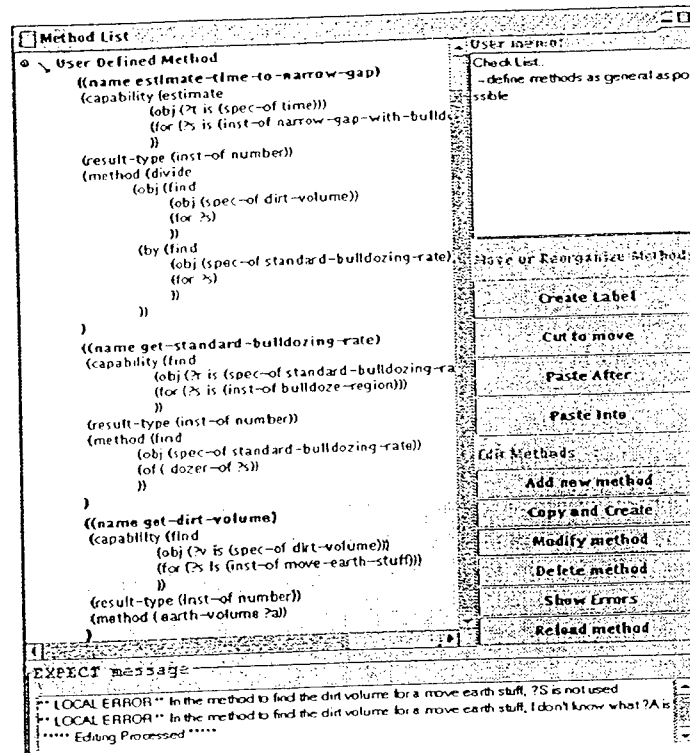
Figure 2: EMeD Interface (Editor).

Finally, once the KA tool indicates that there are no errors, inconsistencies, or missing knowledge, the user can run the inference engine, exposing additional errors in solving a given problem or subproblems. The errors are caused by particular interdependencies among KB elements that arise in specific contexts. If most of the errors are detected b the above analyses, users should see significantly fewer errors at this stage.

Notice that as the KB is more complete and more error-free it becomes a stronger basis for the KA tool in creating expectations to guide the user.

4.4. EMeD: Expect Method Developer

We have concentrated our initial effort in developing a KA tool that uses these kinds of expectations to support users to develop problem solving methods. We built a KA tool called *EMeD* (EXPECT Method Developer) for the EXPECT framework for knowledge-based systems.

60
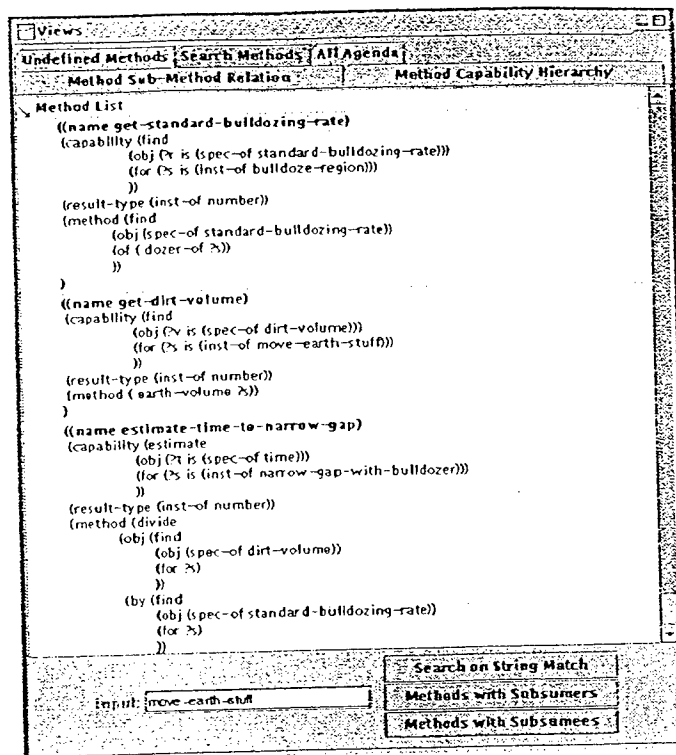
Figure 3: Search Methods in EMeD.

An EXPECT knowledge base is composed of factual knowledge and of problem-solving knowledge. The factual knowledge includes concepts, relations, and instances in Loom (Macgregor 1990), a knowledge representation system of the KL -one family. The problem-solving knowledge is represented as a set of problem-solving methods such as those shown in Figure 2. As described earlier, each method has a capability, a result type, and a method body. Within the capability and body sections, each goal is expressed as a goal name followed by a set of parameters. Also, each parameter consists of a parameter name and a type.

Table 2 shows the method editor in the EMeD user interface. There is a list of current methods and buttons for editing meth ods. Users can add, delete, or modify the methods using these buttons. (Other buttons and windows will be explained later.) Users often create new methods that are similar to existing ones so the tool has a copy/edit facility.

Every time a new method is defined, the method is checked for possible parsing errors based on the method representation language. If there are interdependencies among the subparts of a method, they are also used in detecting errors. For example, if a variable is used but not defined for the method, the same variable is defined more than once, or there are unused variables, the system will produce warning messages. Also, if there were terms (concepts, relations, or instances) used in a method but not defined in the KB yet, error messages will be sent to the developer. When the term definition is obvious, as the verbs used in capabilities, a definition will be proposed by the tool. In Figure 2, the small panel in the bottom left corner with the label "EXPECT message" displays these errors. Using this method definition checker, users can detect the local errors earlier, separating them from other types of errors.

Users can find existing methods related with particular terms in concepts, relations or instances through the   oom ontology. The KA tool can retrieve   methods that refer to subconcepts, superconcepts, or a given concept and let   the user create new methods based on related methods. Figure 3 shows the result from retrieving methods about moving earth. The system was   able to find all the methods in Figure 1, because narrow-gap and bulldoze-region are subconcepts of move-earth.
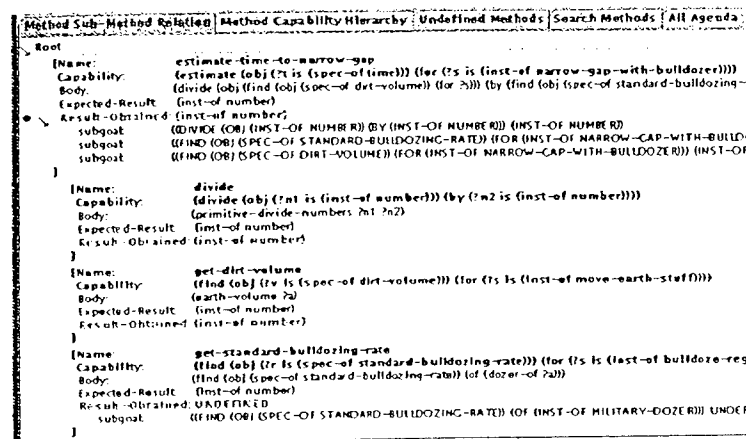

Figure 4: A Method Sub-method Relation Tree.

Figure 4 shows relationships among methods based on how the subgoals of a method can be achieved by other methods. The trees built from this are called *method sub-method relation trees*. There can be multiple trees growing in the process of building a number of methods when they are   not fully connected. These method-relation t rees are incomplete problem-solving trees to achieve some intermediate subgoal. The (sub)trees should be eventually put together to build a problem-solving tree for the whole problem. For example, given these three methods, the system can build a metho d-relation tree, as shown in Figure 4.

Method sub-method relation trees can be used to detect *undefined methods* based on the subgoals in a method that are not achieved by any existing   methods. These can be collected and users can be informed of them. If th ere are constraints imposed on the methods to be built, such as the expectations coming from the methods that invoke them, then these can be also incorporated.   For example, the method to find the standard bulldozing rate of a step calls a subgoal to fi nd the standard bulldozing rate of a given dozer, which is undefined yet. Since the result type of the given method is a number, the system can expect (through an interdependency analysis) the same result type   for the undefined method. Figure 5 (bottom window) shows the capability that the tool proposes

for the currently undefined method --- a method to find standard bulldozing rate of a given military dozer.

In the process of building this method -relation tree, there can be subgoals whose parameters are not fully specified because their arguments are subgoals that are not achieved by any of existing methods. For example, given the method to estimate the time to narrow gap (the first method in Figure 1) only (i.e., if M2 and M3 were missing), its subgoal 'divide' has two parameters with parameter names 'obj' and 'by'. Because the arguments to divide are the subgoals 'find dirt -volume of the step' and 'find standard -bulldozing-rate for the step' whose methods would be undefined, the tool could not full state the goal. This would be represented as 'divide (obj   UNDEFINED) (b UNDEFINED)'. However, one of the built-in methods in EXPECT has   capability of 'divide (obj Number) (by Number)', and the tool creates a link   between this and the subgoal as a *potential interdependency*. Users can   use this hint to make the potential interdependency a real one or create other appropriate methods.
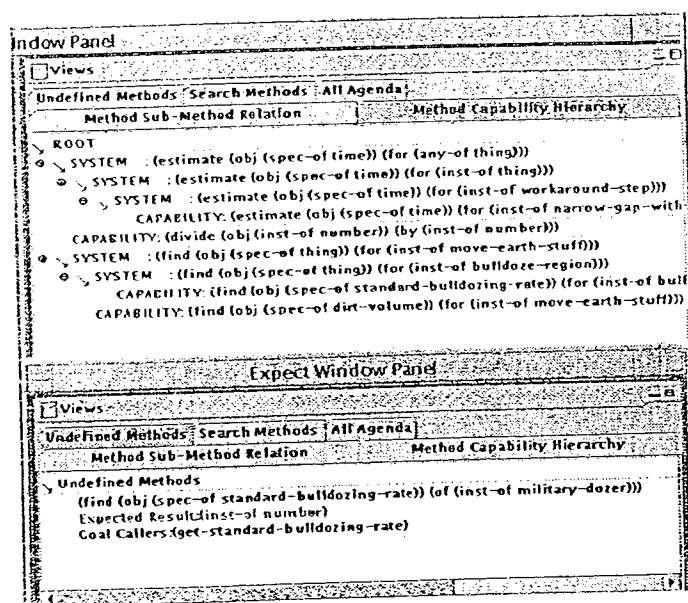

Figure 5: A Capability Tree and Undefined Methods.

There are other relationships among problem-solving methods based on their capabilities that the KA tool can exploit. For example, a hierarchy of the   goals based on the subsumption relations of their goal names and their   arguments can be created. In the hierarchy, if a goal is to build a mil itary bridge, and another goal is to build a kind of military bridge, such as an AVLB, then the former subsumes the latter. This dependenc among the goal   descriptions of the methods (called *capability tree*) is useful in that it

63

allows the user to understand the kinds of sub-problems the current set of methods can solve. To make their relationships more understandable, EMeD also computes potential capabilities. For example if there are super-concepts defined for the parameters of a capability, a capability with these concepts is created as a parent of the capability. The capability tree for the given example methods is in Figure 5 (top window).

Finally, there are user-expected dependencies among the problem-solving methods, which are usually represented in comments or by grouping of methods in the files where they are built. They do not directly affect the system, but they often become the user's own instrument to understand the structure of what they are building. Also, it can be the user's own interpretation of additional interdependencies among methods. EMeD provides a way of organizing methods into hierarchies and groups, and allows users to provide documentation for the methods. In Figure 2, the "Move or Organize Methods" buttons support these functions.

In addition, EMeD can use the expectations derived from running the problem solver, detecting problems that arise while attempting to build a complete problem-solving tree.

4.5. Preliminary Evaluations

We performed a preliminary evaluation of our approach by comparing the performance of four subjects in two different KA tasks from a Workarounds domain. Each subject did one of the tasks with EMeD and the other task using a version of EMeD that only allowed them to edit methods (the buttons to add, delete, or modify methods), but did not have any additional support from EMeD. Before the experiment, each subject was given a tutorial of the tools with simpler scenarios. The scenarios and tools were used in different orders to reduce the influences from familiarity with tools or fatigue. Each experiment took several hours, including the tutorial, and we took detailed transcripts to record actions performed by the subjects. The subjects had some previous experience in building EXPECT knowledge bases, but not with EMeD.

Results	Total Time (min)	Number of Methods Added	Time/Method (min)
Without EMeD	218	25	8.72
With EMeD	24		6.38

Average time savings: 30 %

Table 1: Results from experiments with subjects.

Functionaliy	Number of Times

Undefined Methods	23
Editor Error Message	17
Method Sub-method Relation Tree	7
Capability Tree	10
Search Method	1
Method Organizer	2
Problem Solving Agenda	5

Table 2: Number of times that the different components of EMeD were used.

64

In Table 1, the total time is computed by summing the times with each subject for each tool. The time for each subject is the time to complete the given task (by creating a successful problem-solving tree and eliminating errors). EMeD was able to **reduce the development time to 70%** of the time that users needed without it. Subjects built a comparable number of methods with the different tools. Note that the subjects were not exposed to the EMeD environment before, but were very familiar with the EXPECT framework. The time savings may be more if the subjects had more familiarity with EMeD. We had multiple trial experiments with one of the subjects, with slightl different tasks, and the subject had become more and more skillful, reducing the time per each action. For these reasons, we expect that the time reduction with EMeD will be larger in practice. Also note that there is a practical limit to the amount of time saved using any KA tool. There is a significant amount of time that use rs spend doing tasks such as thinking and typing, where a machine can provide little help. We would like to measure the improvement over the time actually doing knowledge input, instead of the time to complete a KA task.

We counted the number of times each component of EMeD was used during the experiments, as shown in Table 2. The list of undefined methods was most useful (used 23 times to build 24 methods) during the experiment, and the subjects checked it almost every time they created a new method. The subjects seemed to be comparing what the expected with the list created by the KA tool, and built new methods using the suggestions proposed by the system. The error messages showed after editing each individual method effectively detected the e rrors within a method definition, and was used every time there were local errors in the definition.

Users looked at the Method Sub-method Relation Tree but not as many times as what we expected. The subjects felt the tree was useful but there were too m any items shown for each node, making it hard to read. We are planning to display items selectively, showing the details only when they are needed. The Capability Tree was often used to find some capability description of another method needed while de fining a method body. However, the hierarchical structure was not so meaningful to the subjects, since sometimes people choose arbitrary concepts (compute, estimate, etc.) to describe their capabilities. We are planning to develop a better way of organizing the methods based on what tasks they can achieve.

Search Methods and Method Organizer were used very little during the experiment. Since the size of the KB was relatively small (about 3 methods were given in the beginning and 6 methods were built f or each task), the subjects were able to see them easily in the editor window. However, during real KB development, the size of the KB is usually much larger, and we expect that they will be more useful in real settings.

With EMeD, the subjects run the problem solver mostly to check if they had finished their tasks. EMeD was able to find errors earlier and provide guidance on how they ma fix the problem, filtering out most of the errors. Without EMeD, the subjects run the problem solver to detect errors, and ended up spending more time to find the sources of errors and fix them.

4.6. Related Work

There are other KA tools that take advantage of relationships among KB elements to derive expectations. Teireisias uses *rule models* to capture relationships among the rules based on their general structure and guide the user in following that general structure when a new rule is added that fits a model. Some of the capabilities of EMeD are similar in spirit to the rule model (e.g., the method capability tree), and are also used in EMeD to help developers understand potential dependencies among the KB elements. Other KA tools also use dependencies between factual knowledge and problem-solving methods to guide users during knowledge acquisition. These tools help users to populate and extend a system that already has a significant body of knowledge, but they are not designed to help users in the initial stages of KB development. More importantly, these tools are built to acquire factual domain knowledge and assume that users cannot change or add problem-solving knowledge. In this sense, EMeD is unique because it guides users in adding new problem-solving methods.

In the field of software engineering, it has been recognized that it is generally better to focus on improving the process of software development rather than on the output program itself. Our approach embraces this view and tries to improve the initial phases of KB development. Some previous work on using formal languages to sp   ecif knowledge bases is inspired by software engineering approaches. This work provides a framework for users to model and capture the initial requirements of the system, and require that users are experienced with formal logic. Our approach is complementary in that it addresses the stage of implementing the knowledge-based system, and we believe that our formalism is more accessible to users that have no formal training. Other approaches, support users in the initial stages of development by prov iding a methodology that can be followed systematically to elicit knowledge from experts and to design the new system. These methodologies can be used in combination with our approach.

There is also related research in developing tools to help users bui ld ontologies. Unlike our work, these tools do not tackle the issue of using these ontologies within a problem-solving context. Many of the research contributions in these tools concern the reuse of ontologies for new problems, collaborative issues i n developing knowledge bases, and the visualization of large ontologies. We believe that integrating our approach with these capabilities will result in improved environments to support KB creation.

4.7. Summary

We analyzed the process of KB developm ent to support KB creation and KB extension, and found a set of expectations to help KA tools guide users during the development process. We have classified the sources of errors in the KB development process based on their characteristics, and found wa   s to prevent, detect, and fix errors earlier. These expectations were derived from the dependencies among KB elements. Although EMeD aims to provide support for KB creation, its functionality is also useful for modifying existing knowledge or populating a KB with instances.

We are now extending the EMeD framework to be able to derive expectations in solving particular problems. Currently EMeD computes relationship among the KB components regardless of the context. Depending on what problem episode we are solving, the relationships may show different patterns, since the problem-solving methods may become specialized.

In our initial evaluations, EMeD was able to provide useful guidance to users reducing KB development time by 30%. We expect that EMeD will be even more beneficial for domain experts who don't have much KA experience. EMeD also opens the door to collaborative tools for knowledge acquisition, because it captures what KA tasks remain to be done and that may be done by other users.

5. Extending the Role-Limiting Approach: Supporting End Users to Acquire Proble -Solving Knowledge

Role-limiting approaches have been successful for acquiring knowledge from domain experts. However most systems using this approach ask the user a pre-determined sequence of questions and are therefore not flexible enough for a wide range of situations. They also typically do not support acquiring problem-solving knowledge, but only instance and type information. We extend the role-limiting approach with a knowledge acquisition tool that dynamically generates questions for the user based on a background theory of the domain and on the user's previous answers. The tool uses KA scripts to give an overall structure to a session. We show how to augment a background theory based on problem-solving methods to support this style of interaction. We present results from tool ablation experiments with domain experts in two different domains based on a problem solving method for plan evaluation. When the tool was used, tasks were completed in substantially less time than when the ablated version was used. In addition participants were able to complete substantially more tasks on average, twice as many in one of the studies.

5.1. INTRODUCTION

In order to be successful, deployed intelligent systems need to be robust to changes in their task specification. They should allow users to make modifications to the system to control how tasks are performed and even to specify new tasks within the general capabilities of the system. A common direction of work in knowledge acquisition (KA) aims to support this ability through explicit domain-independent theories of the problem-solving process, often called problem solving methods (PSMs). Such theories can encourage re-use across different applications and the structured development of intelligent systems as well as providing a guide for knowledge acquisition from experts. Our interest is in using background theories to guide automatic knowledge acquisition from domain experts who are not necessarily programmers. Background theories of the problem-solving process are appealing for this purpose because their model of the overall structure of the process can in turn be used to structure the knowledge acquisition session for the user and provide context for the knowledge that is acquired. However, most KA approaches that use problem solving methods typically focus on assisting knowledge engineers rather than domain experts.

Other KA tools such as SALT follow a role-limiting approach that allows domain experts to provide domain-specific knowledge that fills certain roles within a PSM. However, most of these have been used to acquire instance-type information only. Musen argues that although role-limiting strategies provide strong guidance for KA, the lack the flexibility needed for KBS construction. The problem-solving structure of an application cannot always be defined in domain-independent terms, and one problem-solving strategy may not address all the particulars of an application because it was designed to be general. Musen and others advocate using finer-grained PSMs from which a KBS can be constructed . Gil and Melz address this issue by encoding the PSM in a language that allows any part of the problem-solving knowledge to be inspected and

68

changed by a user. In their approach, a partially completed KBS can be analysed to find missing problem-solving knowledge that forms the roles to be filled. This work extended the role-limiting approach to acquire problem-solving knowledge and to determine the roles dynamically.

However, Gil and Melz's approach is still not adequate to allow an end user to directl add problem-solving knowledge. There are at least two reasons for this. First, there is no support for a structured interaction with the user provided by tools like SALT. The knowledge roles, once generated, form an unstructured list of items to be added, and it can be difficult for the user to see where each missing piece of knowledge should fit into the new KBS. Second, the user must work directly with the syntax of their language to add problem-solving knowledge, which is not appropriate for end users.

In this chapter we present an approach that builds on Gil and Melz's work to allow an end user to add problem-solving knowledge to a KBS with a dynamic role-limitin approach. In our approach, the tool structures an overall KA session using KA scripts. While following the scripts, background knowledge is used to dynamically generate questions for the user in response to the user's answers to previous questions. As we describe in the next section, this background knowledge includes ontologies that describe the problem-solving knowledge to be added in terms that make sense for the user. Other roles are generated, as before, from an analysis of a partially completed KBS. The background knowledge is also augmented to support interaction with the user through constrained English sentences. We describe an implemented tool, called PSMTool, that uses this approach and show empirically that end users without programming experience are able to add problem solving knowledge to a KBS with this system. The main contributions of this chapter are (1) showing how to use background knowledge to generate a structured dialog with an end user to acquire problem-solving knowledge and (2) demonstrating an implemented KA tool that successfully uses the approach.

We report on two sets of experiments that tested PSMTool. Experts in two different domains where asked to add new evaluation criteria to plan evaluation systems. The KA tasks required providing a combination of problem-solving and ontological knowledge that was integrated with the existing system. In both domains, experts were able to complete substantially more tasks using the KA tool and performed the tasks more quickly.

In the next section we describe a background theory for plan evaluation that we have used in this work. Then we describe how the KA tool uses the theory to generate questions for the user, with examples from a system that acquires new knowledge to evaluate travel plans. We then present results from our experiments and survey related work. Finally we discuss how the results might extend to other background theories of problem solving and mention future work.

5.2. A BACKGROUND THEORY OF PLAN EVALUATION

In this section we briefly describe the background theory for plan evaluation that was used in our experiments and is used in this chapter to illustrate PSMTool. The tool itself

is designed to work independently of a particular theory, and we begin by describing necessary features of a background theory in order for this approach to be successful.

5.2.1. How a background theory supports PSMTool

We use a collection of fine-grained PSMs as the basis for the background knowledge in PSMTool, and add to it the knowledge required for the tool to take over some of the tasks of the knowledge engineer. When using a collection of PSMs to build a knowledg -based system, the knowledge engineer typically selects a component from the collection that is best suited to the task and adapts it to address the task at hand. We support this activity with two additions to the background theory of the task. First, we provide an ontology for the elements of the collection that distinguishes them based on features of the tasks they perform rather than their procedural structure. The ontolog has a hierarchical structure with a number of orthogonal partitions that can be used to generate a dialog with the user dynamically as we describe below. Second, we provide guidance for adapting each element in the collection. Each element is a generic PSM for a given task, and is implemented as a collection of EXPECTmethods. Within each element, we identify the methods that are designed to be changed when the element is specialised. This information is used by PSMTool to further guide the KA process once a PSM element has been chosen. We illustrate these points in the context of a background theory for plan evaluation implemented in the EXPECT system.

EXPECT is an architecture for knowledge acquisition that makes use of an explicit representation for both factual and problem-solving knowledge. It analyses an intelligent system defined in its language to provide KA support to the user, for example b identifying missing problem-solving knowledge or highlighting the information that is needed about objects in the knowledge base. EXPECT's explicit representation makes it ideally suited for representing background theories of problem solving the provide direct support for knowledge acquisition. Previous work has shown that background theories such as propose-and-revise can be represented inEXPECT to take advantage of its analytical capabilities and provide the benefits of both role-limiting KA strategies and completeness analysis.

5.2.2. Plan evaluation

Plan evaluation problems belong to a domain-independent problem class in which an agent, typically a human expert, judges alternative plans according to a number of criteria. The aim is usually to see which of the alternative plans is most suited for some task. In terms of a standard problem solving method library such as common KADS, it is a special case of assessment.
Each different criterion along which the plan is judged is represented explicitly as a critique in our framework. Through experience with several intelligent systems for plan evaluation, Blythe and Gil have identified several patterns in the ways that critiques are evaluated . These patterns of critiques indicate regularities that can be re-used across planning domains and provide guidance for knowledge acquisition.

70

In our background theory for plan evaluation this knowledge is represented through an ontology of critique types, partially shown in figure 1. Some groups of subtypes (shown with arcs connecting their links) partition their parents, while some just specialise the class. For example, the subtypes local-critique and global-critique form a partition, so an critique must belong to exactly one of these classes. local-critique defines critiques that are checked by performing some analysis on some or all of the steps in the plan on an individual basis, while global-critique defines critiques that have a global definition on the plan, for example the plan's total fuel consumption.
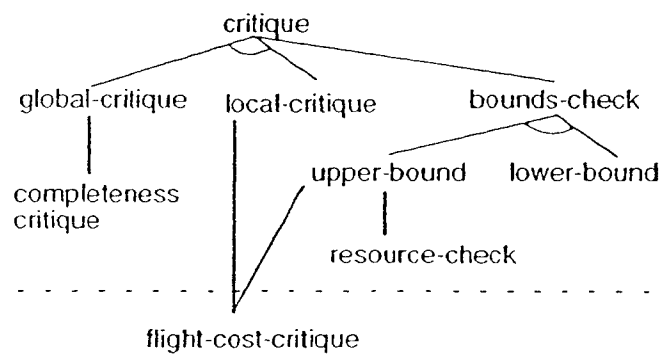
Figure 1: Different types of critiques in the background theory of plan evaluation. Each subclass is identified with a recurring pattern for evaluating a critique, implemented through EXPECT methods attached to the class.

In addition to the ontology of critiques, the background theory uses an ontology of planning terms and a specialised ontology of resources based on Ozone. More details about these can be found in.

The critiques are defined operationally through EXPECT methods. For example, the method in Figure 2 says that a step satisfies an upper bound (a kind of critique) if and only if the actual value of the property to be measured by the critique is less than or equal to some maximum value. The tasks of estimating the actual and maximum values for the property respectively form two subgoals that are addressed by other methods in the theory.

```
(define method evaluate-local-upper-bound
    "To determine whether a step satisfies an upper
bound, check that the actual value of the property
to measure for the upper-bound is less than or equal
to the maximum allowed value of the property."
    :capability
       (determine-whether                     .
            (obj (?task is (inst-of step)))
            (satisfies (?bc is (inst-of upper-bound))))
    :result-type (inst-of boolean))
    :method
       (is-less-or-equal
           (obj (estimate (obj actual-value)
                          (of (property-to-measure ?bc))
                          (for ?task)))
           (than (estimate (obj maximum-allowed-value)
                       (of (property-to-measure ?bc))
                          (for ?task)))))
```

Figure 2: An EXPECT method.

The background theory can be used to create a working plan evaluation system for a
particular domain by defining domain-specific critique classes within the ontology and
adding the necessar EXPECT methods to complete each critique's definition. We
illustrate this in a travel planning domain. In this domain, plans are itineraries for travel
and the steps in plans represent reservations of flights, hotels and rental cars. The plan
critiquer for the travel planning domain is implemented as an extension of the
background theory for plan critiquing. For example, one possible critique in the domain
checks that no flight costs more than a set amount, say $500. This is implemented b
defining the class flight-cost-critique as a subclass of both local-critique and upper-
bound. The value of the relation propert -to-check is required for any subclass ofupper-
bound, and here it takes the value flight-cost. The EXPECT methods for those classes are
then used to evaluate the new critique, resulting in a check that the actual amount of
flight-cost for each step is less than or equal to the maximum amount. The developer
completes the definition by defining methods to compute the actual amount (the cost of
the flight) and the maximum amount ($500)

5.3. INTERACTING WITH PSMTOOL

We have implemented a tool, called PSMTool, that makes use of a background theor
for problem-solving to guide a user in extending the capabilities of an intelligent system.
PSMTool follows a script to organise its interaction with the user that is hand-written for
the background theory. However, the questions that the tool asks the user while it
executes the script are automatically derived from the theory, as we show below. Before
describing this we show an example of the tool's interaction with a user.
                                              .

The KA tool begins with a window split in two screens, showing a plan on the left and on the right a list of evaluations of the plan made with the critiques currently in the knowledge base. The user can choose to add a new critique, in which case a second window appears that will keep track of the interactions with the user made in defining the critique, shown in Figure 3. The interaction follows a script that has three steps: first, it adds a new critique class and classifies it in the ontology, then it generates a list of EXPECT methods for the user to edit that will be used in evaluating the critique. Once these steps are complete the new critique has been defined. The third step then applies the critique to the plan in the first window and the result is shown for the user to verify. We now show how the questions used in the first two steps of the script are dynamicall created.

Figure 3: The critique window asks the user questions in order to classify the new critique within the background ontology and then prompts the user to modify several new EXPECT methods to tailor the behaviour of the new critique.

73

5.4. SUPPORTING KNOWLEDGE ACQUISITION WITH BACKGROUND THEORIES

We are interested in using a background theory to provide support for users who are not necessarily programmers to extend and tailor systems for their domains. We added the following kinds of information to the background theory described above.

The critique ontology, which records functional differences in types of critique based on their patterns of usage, can be used to generate questions to get information from the user about a new critique without requiring programming knowledge. To do this, each class in the ontology is augmented with a text description that characterises the class and, in the case of partitions, distinguishes the class from other elements in the partition. In many cases, once the user has provided information so the new critique is classified, much of the problem-solving knowledge needed to implement the critique is available through the critique's parents in the background theory. This problem-solving knowledge can handle overall organization of the critique and its integration with the system, which is sometimes the most complex part of the critique's procedural definition. The background theory also distinguishes some EXPECT methods as default methods that may be specialised when a new critique is defined. This serves two purposes. While all the methods in the background theory can in principle be specialised, the distinguished subset can give the KA tool a road-map for tailoring the theory for a new critique. In addition, the body of the general method can provide a starting-point for the specialised method and the general method can be included for this reason, even if it is not intended to be used without specialisation.

5.4.1. CLASSIFYING THE CRITIQUE FROM THE ONTOLOGY

The questions asked in the first part of the critique window are automatically derived from the ontology of critiques in the theory and classify the new critique according to the the ontology. Once the classification process is complete, a new critique class is defined for the new critique as a subclass of the identified classes. This class is able to inherit generic critiquing knowledge from its ancestors as described in the previous section. The algorithm for determining questions in the critique script adds each question for one of three reasons: 1) to determine which element of a partition the critique belongs to, 2) to determine whether the critique is a subclass of some class that is not part of a partition and 3) to find the value of a required field of a class. The algorithm traverses the critiques in depth-first order to help maintain a focus of attention for the user in answering the critiques. Figure 4 shows the algorithm that generates the questions in the critique window.

74

Initialise C to the partitions and subclasses of critique
1 if C is empty, stop
2 set c to the first element in C, and remove it from C
3 if c is a partition of subclasses,
4 ask which subclass in c the new critique belongs to.
5 set n to the answer
6 else if c is a subclass,
7 ask whether the critique belongs to the subclass
8 if YES, set n to c
9 if NO, goto step 1.
10 for each required field r of n:
11 ask for the value of r.
12 push the partitions and subclasses of n onto C
13 goto step 2.
Figure 4: The algorithm that dynamically generates classification questions in the critique window.

Figure 3 shows the questions that are generated using this algorithm for the class flight-cost-critique, defined as shown in Figure 1. The text for each question is stored with each class or required field in the ontology. Although the algorithm requires the developer to store canned text with the classes and fields in the ontology, this is considerably easier than generating scripts of questions for the different possible classifications of a critique. The answers to the five questions in steps one through five have allowed the KA tool to classify the new critique class as a subclass of both upper-bound-check and local-constraint.

5.4.2. CHOOSING AND CREATING NEW METHODS TO EDIT

Once the new critique has been classified in the ontology, the system builds a problem-solving tree in EXPECT for the generic goal to determine whether a plan satisfies the critique. EXPECT is able to detect subgoals for which no method can be found and the information that is required of objects in the domain, such as plans and critiques. Requesting information that is missing in these categories is similar to the behaviour of role-limiting knowledge acquisition systems such as SALT, but more flexible since it is generated from an analysis of an explicit model of the background theory. This connection is discussed in more detail in. We augment this algorithm so that it also finds subgoals addressed by methods that are marked in the background theory as specialisable. The KA tool then prompts the user to enter methods for these subgoals one by one. If a specialisable method exists, this forms the initial version of the new method.

The user edits problem-solving methods via the English-based editor, which uses constrained English sentences to present and edit the methods rather than the formal language in which the methods are defined. The method capability and body are presented as structured English, and when the user selects a word or group of words in them, the lower window of the editor shows alternatives that could be substituted for the

selection and maintain the syntactic correctness of the method. The alternatives can include Loom or EXPECTexpressions. This editor is important to the success of PSMTool, since the intended users are not programmers and would not be comfortable with formal languages for ontologies or problem-solving knowledge.

In the case of the class flight-cost-critique, the problem solving tree makes use of the method in Figure 2, and the critique tool finds three specialisable methods, which are shown in the lower part of the critique window in Figure 5. Figure 3 shows the English editor being used to define one of the methods, to compute the cost of a flight. Once these three methods are defined, the new critique is ready to be applied. In this case, they can be defined using just Loom expressions and constants, without calling other EXPECT methods, although the editor allows this.
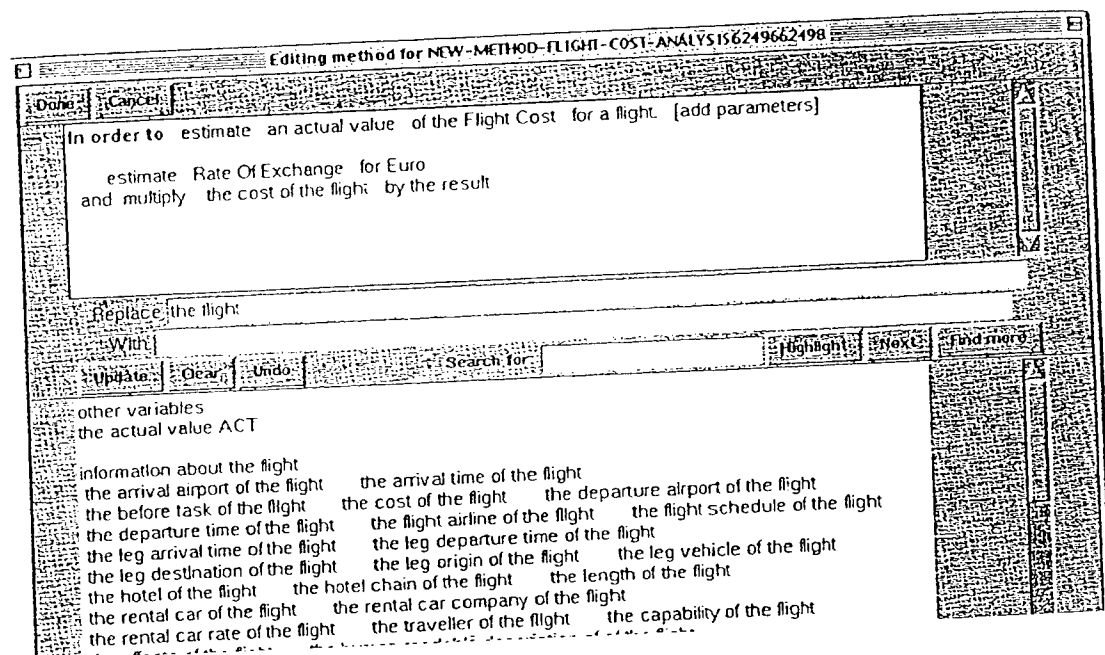

Figure 2: The English-based method editor in use to change the method to compute the actual cost of a flight.

5.5. EXPERIMENTAL RESULTS

Our aim is to design a knowledge acquisition tool that can be used by domain experts. To test the tool, we ran two sets of experiments that test the same hypotheses, but in different plan evaluation domains and with different sets of domain experts. The hypotheses we tested are 1) that with the knowledge acquisition scripting tool, domain

experts are able to add more complex critiques than without the tool and 2) that domain experts are able to add the same critique more quickly using the tool. In each domain, we ran a tool ablation experiment, in which participants used two versions of a knowledge acquisition environment that were identical except that only one gave access to the KA tool. Both versions allowed users to add a critique class through a class browser and to start the English method editor to add a method.

Participants were asked to define new critiques for a plan evaluation system. In order to control the conditions of the experiment as far as possible, the critiques to be added using the tool were structurally isomorphic to the critiques to be added with the ablated version. In other words, corresponding critiques from the two groups critiques required the same pattern of EXPECT methods and loom retrievals in their definitions, although the names of the methods and loom relations could be different. In order to control for learning effects during the course of each experiment, half the participants worked first with the tool and then with the ablated version, and half worked the other way around. The participants in the first experiment were four Army officers from the Army Battle Command Battle Labs in Fort Leavenworth. The problem domain for this experiment evaluated Army battle plans. The participants in the second experiment were three project assistants from our division and one business school professor, and the problem domain was the travel plan evaluation system that we have used to illustrate the chapter. None of the participants from either experiment had written a computer program. It is unfortunate that the number of participants was too low to draw statistically valid conclusions from these experiments, but the experiments are costly to run since the require one experimenter per participant to make observations.

The critiques in each set were graded in difficulty, so that we could also investigate the effect that this had on the impact of the KA tool. The simpler critiques can be implemented with one EXPECTmethod, while the more complex critiques require iterating over a set of steps in the plan, or require two or three EXPECT methods to be specialised if they are defined using the background ontology.

As an example of a simple critique, participants in the second study were asked to define a check that the plan included a hotel reservation. As an example of a more complex critique, participants in the first study were asked to define a check that the force ratio, defined as the ratio of friendly fire-power to enemy fire-power, is adequate for each task in the plan. The level that is considered adequate must be computed based on the task. Figure 6 shows the number of KA tasks that were completed by the subjects with the KA tool and with the ablated version in each of the domains. In this figure, each critique has been split into a number of component tasks, corresponding to adding a new critique class or completing either the capability or body of a method. We have aggregated the tasks from the two different sets given to the participants. As the graph shows, use of the tool does not affect the number of tasks that are completed for the easiest critique, but there is a marked difference for more complex critiques. The same is true if we consider the number of whole critiques added rather than the component tasks. Participants in the battle planning experiment were able to complete more than twice the tasks with the tool than were completed with the ablated version. Participants in the travel planning experiment showed smaller increases but still significant ones.
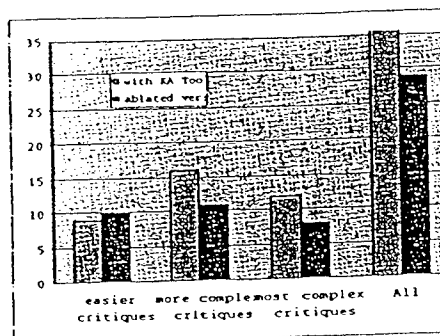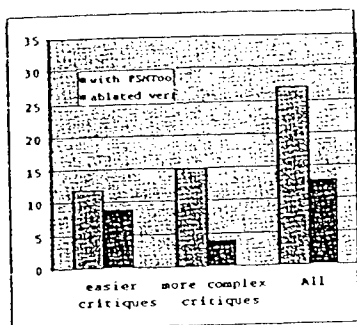
Figure 6: Tasks completed with the full tool and with the ablated version. Results in the battle planning domain are shown on the left and results in the travel planning domain are shown on the right.

Figure 7 shows the average times taken to define a critique in each domain. To ensure that the times are comparable between the standard and ablated versions of the system, only participants who completed both tasks were used to compute the averages. The point representing the most complex critique attempted without the KA tool is missing from both experiments because no participant successfully added the critique.
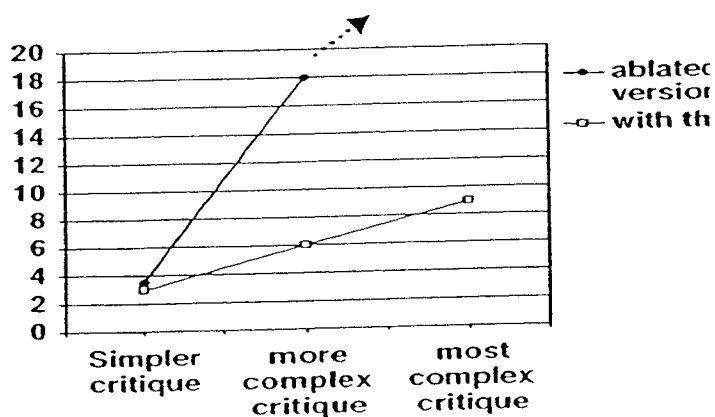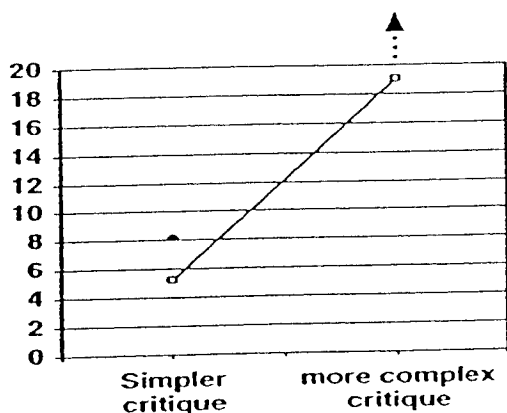
Table 7: Average times (in minutes) taken to define each critique. Results from the battle planning domain are shown on the left and results from the travel planning domain are shown on the right.

The results from these experiments support our hypotheses. Participants complete significantly more KA tasks using the tool, and take considerably less time to do it. One interesting difference between the participants in the two experiments was the amount of training that they received for the task. The battle planning experiments were

78

part of a number of experiments made at Fort Leavenworth with different KA systems and the participants received eight hours of training before using the tool --- four hours of training with EXPECT and four hours with a related KA tool. In the travel planning experiments, participants received only one hour of training with the tool and none with EXPECT. However the number of tasks completed and the average times are ver similar between the two groups. This indicates that domain experts might require ver little training to be able to use the tool. We have not yet studied this systematically, but can offer one more anecdotal piece of evidence: at Fort Leavenworth, a fifth participant was asked to add the same critiques using the KA tool after less than one hour's training. The participant succeeded, and took time comparable with the participants who had received eight hours of training.

5.6. RELATED WORK

Other knowledge acquisition tools help users add domain-specific information to a knowledge base. Some, such as ASTREE use explicit ontologies as we do to organise problem-solving methods. Like our tool, these tools are built for domain experts and not knowledge engineers, but they do not allow users to provide domain-specific problem-solving knowledge as PSMTool does.

Other researchers have developed tools to support the acquisition of planning knowledge, but these systems are built for knowledge engineers. Both of them address some aspects of the acquisition of plan evaluation knowledge, but their central focus is plan generation. Valente and his colleagues have also studied problem-solving methods related to planning. Our model has similarities to theirs but is more aimed at plan evaluation, a separate activity from plan creation that has been less studied from a computational viewpoint. Another difference in our approach is the automatic support for KA via EXPECT and script generation.

5.7. SUMMARY

We described an extension to the role-limiting approach for knowledge acquisition from end users that dynamically generates knowledge roles to be filled and can be used to acquire problem-solving knowledge. Our approach combines a collection of problem-solving methods with supporting ontologies that can be used to guide the activities of method selection and specialisation. As well as demonstrating the approach for a plan evaluation problem solving method, we have shown through controlled experiments that domain experts can provide more complex knowledge to an intelligent system using PSMTool and that they take less time to provide the same knowledge.

Although we have discussed these techniques in the context of the plan evaluation theor they are of course not limited to this background theory. The ways in which we augment the theory to support knowledge acquisition would be useful for a range of theories, and our algorithms for generating questions to ask a user are domain-independent. The script that we demonstrated, however, was specific to the plan evaluation theory. Other background theories would certainly need their own scripts and will probably need a greater variety of ways to support knowledge acquisition. Future work includes

generalizing this result to other classes of task. We plan to investigate theories to help acquire process control knowledge and search control knowledge for AI planning systems, among others.

6. User Studies of Knowledge Acquisition Tools: Methodology and Lessons Learned

The area of knowledge acquisition research concerned with the development of knowledge acquisition (KA) tools is in need of a methodological approach to evaluation. Efforts such as the Sisyphus experiments have been useful to illustrate particular approaches, but have not served in practice as testbeds for comparing and evaluating different alternative approaches. This chapter describes our experimental methodology to conduct studies and experiments of users modifying knowledge bases with KA tools. We also report the lessons learned from several experiments that we have performed. Our hope is that it will help others design or improve future user evaluations of KA tools. We found that performing these experiments is particularly hard because of difficulties in controlling factors that are unrelated to the particular claims being tested. We discuss our ideas for improving our current methodology and some open issues that remain.

6.1. Introduction

The field of AI has increasingly recognized throughout the years the need and the value of being an experimental science. Some subfields of AI have developed standard tasks and test sets that are used routinely by researchers to show new results. Researchers in machine learning, for example, use the Irvine data sets to show improvements in inductive learning and routinely use tasks like the n-puzzle or the n-queens for speed-up learning research.

Developing standard tests is harder in other subfields that address more knowledge-intensive problems. For example, planning researchers often show experiments in similar task domains. The problem is that the implementation of the knowledge base and of the algorithms is so different across systems that the results of the experiments are often hard to analyze. One approach used by some researchers is to use artificially created, very structured knowledge bases to analyze particular behaviors. Another approach has been to define a universal language to describe planning domains, as is done in the Planning Competition of the Artificial Intelligence Planning Systems Conference.

As developers of knowledge acquisition (KA) tools we wanted to evaluate our approaches, and began looking into user studies. With the exception of some isolated evaluations of KA tools we found that the field of knowledge acquisition had no methodology that we could draw from to design our evaluations. Even though AI is, as we mentioned earlier, a field where experimental studies have been increasingl emphasized in recent years, user studies are uncommon. User studies to evaluate software tools and interfaces can be found in the literature of tutoring systems, programming environments, and human computer interfaces. These communities are still working on developing evaluation methodologies that address their specific concerns. All seem to agree on the difficulty and cost of these studies, as well as on their important benefits. Often times, the evaluations that test specific claims about a tool or approach are perhaps not as thorough or conclusive as we would like to see as scientists, yet we are lucky that these evaluations are taking place at all and are shedding some light

on topics of interest. In developing a methodology for evaluation of KA tools, we can draw from the experiences that are ongoing in these areas.

The lack of evaluation in knowledge acquisition research is unfortunate, but may be due to a variety of reasons. First, KA evaluations are very costly. In areas like machine learning and planning, experiments often amount to running programs repeatedly on already existing test sets. The evaluation of a KA tool requires that a number of subjects spend a fair amount of time doing the study, and for the experimenters to spend time and other resources preparing the experiment (often months) and analyzing the results. Second, most of the research in the field of KA concentrates on knowledge modeling (e.g., how a knowledge engineer models a task domain) and knowledge elicitation (e.g., techniques for interviewing experts). There are very few efforts on developing tools for users. Developers may conduct usability studies, but the results are not reported in the literature.

In recognition of the need to evaluate KA research, the community started to design a set of standard task domains that different groups would implement and use to compare their work. This effort is known as the Sisyphus experiments and the domains have included office assignment, elevator configuration, and rock classification. These experiences have been useful to illustrate particular approaches, but have not served in practice as testbeds for comparing and evaluating different approaches. The most recent Sisyphus is an example of the issue discussed above about the intimidating cost of KA evaluations: the limited number of participants can be tracked back to the significant amount of resources required to tackle the knowledge-intensive task that was selected. Over the last few years, we have performed a series of evaluations with our KA tools that have yielded not only specific findings about our tools but that have also allowed us to develop a methodology that we follow in conducting our evaluations. This chapter describes our experimental methodology to conduct studies of users modifying knowledge bases with KA tools. It also reports the lessons learned from our experiments so it will help others design or improve future user evaluations of KA tools. This chapter describes our experiments in enough detail to illustrate the different point of our methodology and the lessons learned. A more comprehensive description of our experiments and their results can be found in the literature.

The chapter describes our experiences based on tests with two particular KA tools that we developed for EXPECT. EXPECT is a framework for developing and modifying knowledg based systems (KBSs) whose main purpose is to enable domain experts lacking computer science or artificial intelligence background to directly manipulate the elements of a KB without the mediation of a knowledge engineering. The two tools that were the subject of our evaluation were intended to enhance some aspect of the EXPECT support to end users. ETM (EXPECT Transaction manager) uses typical KB modification sequences (KA Scripts) to help users modify KBs. EMeD (EXPECT Method Developer) analyzes and exploits interdependencies among KB elements to guide users in making significant KB extensions or changes. Each tool was developed to investigate a different approach to guide users in knowledge acquisition tasks. The approaches are complementary, and we are now in the process of integrating the features of the tools that we found useful in the experiments in order to create a more comprehensive and powerful KA environment for EXPECT. A brief overview of both tools can be found in Appendix A. Please note that the focus of this chapter is not on the

details of these tools but on our experimental methodology and the lessons learned from our experiments.

The chapter begins by describing the methodology that we follow to design experiments to evaluate KA tools, illustrated with examples from our evaluations with ETM and EMeD. The next section highlights the lessons that we learned in carrying out these experiments, and describe open issues in KA experiment design. Finally, we discuss related work in other research disciplines that conduct user studies and outline directions for future work.

6.2. A Methodology to Conduct Experimental User Studies with Knowledge Acquisition Tools

The nature of an experiment is determined by the claims an d hypotheses to be tested. Based on the hypotheses to be tested, we need to determine the KA task to be performed by the users, the underlying representations for the knowledge acquired by the tool, the type of users involved, the procedure to be followed to perform the experiment, and the data that needs to be collected. This section discusses each of these issues in detail.

Table 1 summarizes the steps in our methodology. It is by no means a strictly sequential process, rather there is significant iteration and backtracking across these steps due to the interactions among all the constraints and decisions involved. For example, a hypothesis may be revisited if an experiment cannot be designed to test it as it is stated. It is extremely useful to run a pre-test using a smaller-scale or a preliminary version of the experimental setup (e.g., fewer users), so that the design of the overall experiment can be debugged, refined, and validated.

6.2.1. Stating Claims and Hypotheses

Claims and hypotheses play a pivotal role in the evaluation process, since the experiments revolve around them. Claims and hypotheses are related but not necessarily the same. Claims are stated in broader terms, referring to gen eral capabilities and benefits of our tools. It may or it may not be possible to test a certain claim, but it helps us understand what we think are the advantages of a certain approach. Based on these broader claims, we formulate specific hypotheses that we would like to test. In contrast with claims, hypotheses are stated in specific terms, and we formulate them such that an experiment can be designed to test them and yield evidence that will lead to proving or disproving specific hypotheses. In reality, some experiments are potentially possible but turn out to be infeasible in practice due to lack of time and other resources.

The first step in the design of our evaluations was to state the main claims regarding our KA tools. It turns out th at we made similar claims for both tools:

1. Users will be able to complete KA tasks in **less time** using the KA tools.
   Rationale: Our KA tools would support some time consuming activities involved in KB modification tasks. For example, they support the analysis of the interactions among the elements of KB and the determination of the actions that should be taken to remove inconsistencies in the KB.

2. Users will make **less mistakes** during a KA task using the KA tools.
   Rationale: Our KA tools high light KB inconsistencies and provide support in fixing them.

3. The reduction in completion time and number of mistakes will be more noticeable for **less experienced users**.
   Rationale: Less experienced users will be the most benefited from the tool's thorough guidance. In addition these users would be able to resolve the conflicts that arise during the modification of the KB using strategies that they might not be aware otherwise.

4. The reduction in time will also be more noticeable for **users lacking a detailed knowledge** of the KBS implementation.
   Rationale: Our tools reveal the existence of KB elements that can be reused or adapted and that the users may not be aware of. This should be particularly noticeable when the KB is large.

5. The KA t ools will be useful for a **broad range of domains and knowledge acquisition scenarios**.
   Rationale: Our tools are based in general domain-independent principles.

Given these claims, we were able to state specific and measurabl hypotheses to be proved or disproved with experiments that were feasible give our resources and constraints.

For example, a specific hypothesis for ETM corresponding to claim 1 is: Completion time for a complex KBS modification will be shorter for subject using ETM in combination with the EXPECT basic KA tool than for subjects using the EXPECT basic KA tool alone.

A claim can be stated in more general or more specific terms depending on the purpose of the claim. The claims that we showed above are specific to particular KA to ols and methodologies, but it would be make them part of more general claims that the whole KA field cares about and that other researchers may want to hear about the state-of-the-art in KA. For example, our experiments and those of others might help u s gather evidence towards general claims such as *"It is possible for naive users to make additions and changes to a knowledge base using KA technology"*, with more specific claims stating what technologies help in what kinds of KA tasks to what kinds of users.

6.2.2. Determining the set of experiments to be carried out

It is useful to test one or more hypotheses in few experiments, but it is not always possible. This is the case when the h ypotheses are of a very different nature, or when a

given hypothesis needs to be tested over a range of user types, tasks, or knowledge bases. For example, if we have two different hypotheses, such as (1) a KA tool helps to perform a task more efficiently and (2) the KA tool scales up to large and realistic applications, then it might be necessary to conduct one experiment to support the first hypothesis and a second experiment for the second hypothesis.

A useful way to design an experiment that establishes the benefits of a tool or a technology is to perform a comparison with some baseline tool. In this case, we have to carefully choose the tools to be compared. The only difference between the two tools to be compared has to be the presence or absence of the technology to be evaluated. Otherwise, we may not be able to determine if the differences in the performance of the tools were due to the technology itself or to some other factors (e.g., a different interface design or interaction st le). For example, to test the benefits of expectation-based KA, we compared EMeD against a basic KA tool that consisted of the same EMeD interface where a number of features had been disabled. That is, both tools (EMeD and the Basic EMeD) provided a similar user interface environment.

This kind of experiment is a tool *ablation* experiment, where the object of ablation is some capability of the tool. The group of subjects that is given the ablated KA tool serves as the *control group*. We have found these experiments to be the most useful and compelling kind to test for our claims.

6.2.3. DESIGNING THE EXPERIMENTAL SETUP

Once we have determined the hypotheses and the kind of experiment to be carried out, we are able to plan the details of the experiment.

6.2.3.1. Users

An important issue is the choice of subjects who are going to participate. Practical concerns often constrain the experimentation possibilities, for example the accessibilit and availability of certain types of subjects.

We design our experiments as *within subject* experiments. This means that each subject uses both the ablated and the non-ablated version of the KA tool (but not to do the same task, as we describe below). Because of the many differences among subjects and the small amount of subjects that we could test, this helps to reduce the effect of individual differences across users. Different subjects use the two versions of the tool in different orders, so as to minimize the effects that result from increasing their familiarity with the environment that we provide. Another possibility is to use a larger number of subjects, and divide them into two separate groups: one that uses only the ablated tool and the other that uses the non-ablated tool. The problem with this design is that it is more costly.

We have identified several types of users according to their background and skills, particularly with respect to their familiarity and expertise with knowledge base development and knowledge acquisition tools and techniques.

6.2.3.2. Domains, KA Tasks and Scenarios

One issue is the choice of the application *domain*. For our purposes, the knowledge bases needed to be simple enough to be learned in a short time but complex enough to challenge the subjects in the control group. For ETM, we chose a transportation planning evaluation system developed as a part of a Defense Advanced Research Project Agency (DARPA) funded project. An example of a KA task given to the subjects is to modify a fragment of an existing knowledge-based system capable of evaluating transportation movements carried only by ships in order to enable the system to evaluate movements involving both ships and aircraft. This modification required to add new problem-solving knowledge to handle aircraft and to integrate this new knowledge with the knowledge that already existed in the knowledge-based system.

For EMeD, we chose a Workarounds domain selected by DARPA as one of the challenge problems of the High-Performance Knowledge Bases program that investigates the development of large-scale knowledge based systems. The domain task is to estimate the delay caused to enemy forces when an obstacle is targeted by reasoning about how they could bypass, breach or improve the obstacle. An example of a KA task given to subjects is to add problem-solving knowledge to "estimate the time to move military assets by enemy units", considering the source of the assets, current location, destination, and their moving speeds. For each task, the subjects added new problem-solving knowledge to the system.

There is a range of *difficulty of KA tasks* in terms of the kinds of extensions and/or modifications to be done to a KB. A KA task that requires only adding knowledge is very different in nature and difficulty from a KA task that requires modifying existing knowledge. Also, modifying problem-solving knowledge is a very different task from adding instances, even if they are both KA tasks. It is important to design the scenario so that it covers the kind of KA tasks that the tool is designed for. Both our tools are targeted to the acquisition of problem solving knowledge. We tested ETM with a KB modification task, and EMeD with a KA task that required extending an existing KB b adding new knowledge.

Another issue in within subject experiments is that if one gives a subject the same exact KA task to do with the two versions of the tool there will most probably be a *transfer effect*. This means that it is pretty certain that they will be unlikely to repeat errors the second time they do the task, and that they will remember how they did something and will not need to figure it out the second time around. To avoid this transfer effect, we design two different but comparable *scenarios*, each involving the same kind of KA task in the same domain but involving a different aspect of the knowledge base. One scenario is carried out with the ablated tool and the other one with the non-ablated

version of the tool. It is very important that both scenarios are as comparable in size and complexity as possible in order for the results of the experiments to be meaningful. This results in *four subject groups*, each with a specific combination of the two versions of the tools and the two KA scenarios to be carried out.

To facilitate the subject's acquisition of the knowledge required to perform the scenarios and to ensure a uniform understanding of the domain across users, the application domain was explained to all the subjects during a presentation. The subjects and the domain were especially chosen to avoid markedly differences in the subjects previous exposure to the domain. Of course, these points are not relevant when testing subjects that are experts in a specific domain.

A repository of knowledge bases and scenarios to test KA tools that could be shared b different researchers would enable better comparative evaluations among approaches, as well as reduce the amount of work required to evaluate a KA approach. The knowledge bases that we used are available to other researchers by contacting any of the authors.

6.2.3.3. Experiment procedure

After we had determined the kind of experiment to be carried out, our hypotheses, the type of users, and the nature of the KA task, we were in condition to plan other details of the experiment. These include, for example, what information will be given to the subjects and in what format, what kind of interaction can the subjects have with the experimenters during the tests (e.g., can they ask additional questions about the domain and/or the tool), how many iterations or problems will be given to each subject and in what order, and an indication of the success criteria for the subjects so they know when they have finished the scenario they were given (e.g., the final KB correctly solves some given problems, perhaps according to a gold standard).

The experimental setup has to be carefully designed to control as much as possible the variables that can affect the outcome of the experiment and that are not related to the claims. For example, if the disparity of subjects may affect the results we can have each subject to perform two tasks, one with the tool to be evaluated and the other with the tool used for control. In this case, if the order in which the tasks are executed might also affect the results, we can switch the order in which the tasks are executed for different subjects. If we suspect that not all subjects have the sufficient skills to perform the requested task we can include a practice session previous to the evaluation. We can also ask the subjects to fill a background questionnaire previous to the execution of the experiment to form the groups as balanced as possible.

Our controlled experiments compared the performance of subjects using EXPECT, our baseline KA tool, vs. subjects using enhanced versions of EXPECT that incorporate the tools being evaluated. The subjects that used EXPECT alone constituted the control group. Each subject performed two different scenarios, one with the basic EXPECT and the other with the enhanced EXPECT, so the results were independent of differences in subject's background and skills. Each scenario was performed with both versions of EXPECT so the results were independent of the complexity of the scenario. Some subjects used the plain EXPECT first and others used the enhanced version of EXPECT

first so the results were independent of the order in which the tools were used. All these factors were balanced such that they would even out the qualifications of the subjects for each one of the groups, the number of times that each tool was used for each scenario, and the number of times that each tool was used in the first place.

All experiments followed the same general procedure distributed in two

Stage 1: Domain and tools presentationendtrivlist
The subjects attend a presentation that introduced EXPECT, ETM or EMeD, and the application domain.

Stage 2: Practice and executionendtrivlist
The subjects perform the following activities:

1. Execute a practice scenario comparable to the ones to be used during the actual test. This scenario was performed once with each version of EXPECT. The .purpose of this practice is to make subjects familiar with the tools, the domain, and the procedure of the experiment

2. Execute two test scenarios, one using each version of EXPECT, alternating the order of the tool that is used first.

3. Answer a feedback questionnaire regarding their impressions and difficulties in using each version of EXPECT. Each question is given numerical range (1 to 5), so that the answers are comparable across subjects.

For each test scenario, the subjects analyze the domain and the specifications. Then, the perform the given modification scenario until the KBS gave the correct results in the sample problem. During the execution of the scenarios, the experimenter only occasionally assists subjects that had problems interpreting the instructions, using the KA tools, or that get stuck or confused.
We use several approaches to determine when a subject has completed a KA task appropriately. In most cases, we take advantage of the formal validation mechanisms in EXPECT. In these cases subjects are asked to complete a KA task and make sure that EXPECT does not report any inconsistency. In some other cases, the subjects are asked to test the KBS with a given set of problems, and make sure they obtain the expected results. In addition, after each experiment, we check the modifications made by the subjects by hand.

6.2.4. DETERMINING WHAT DATA TO COLLECT

The kind of data collected during the experiment may be determined and/or limited b what is possible in terms of *instrumenting the KA tool and the KB environment.* Also, intrusive ways of recording data should be avoided. For example, we should not ask the

users to fill a long form to describe what they just did if that is going to disrupt their train of thought and make the overall time to complete the task longer.

The following data was collected during the execution of our scenarios:

- Time to complete the whole KA task.

- Automatic log of changes performed to the KB (e.g., a new parameter was added to a goal).

- Automatic log of errors in the KB after each change to the KBS (e.g., a problem solving goal cannot be achieved). These errors are detected automatically by the EXPECT framework and hence are available in both the basic and the enhanced KA tool.

- Automatic log of the features of the KA tool used during the experiment (e.g., the highlighting of unachievable goals).

- Detailed notes of the actions performed by the subjects (taken manually by the experimenters) including how they approached the problem and what materials they consulted. For this purpose we ask the subjects to voice what they are thinking and doing during the execution of the scenario. We do not use video cameras and tapes, since we find the notes to be sufficient and more cost-effective.

- Questionnaires that the subjects fill out at the end of the experiments, with questions regarding the usability of the tools

The data collected should be sufficient to measure our hypothesis and confirm whether they stand or not. Collecting fine grained data is very useful because it not only proves/disproves the hypotheses, but it also helps to explain the outcome of the experiment, and to explore the potential causes of certain experiment outcomes. For example, the analysis of the sequence of changes performed to the KB revealed that some subjects spent considerable amount of time correcting their own mistakes.

We find that conducting pre-tests is very useful not only to help refine the actual evaluation setup but also the data collection strategy.

6.2.5. Analyzing the Data

To illustrate this step, we show some data from the experiments with ETM and EMeD, and the conclusions that we extracted from them.

Figure 1 compares the subjects performance at each step of the modification for one of the scenarios of ETM. A detailed analysis of this figure reveals some anomalies that might have affected the results. For example, this figure shows that Subject 3 in the control group spent some extra amount of time analyzing incorrect results and fixing errors caused by his/her own mistakes (7 minutes in Change 4 and 1 minute in Change 5). Recovering from mistakes takes a significant amount of time. Hence, it seems unfair to compare the performance of subjects when some of them made mistakes that were not related to the use of ETM. However, even without considering this extra time the time spent by Subject 3 does not come close to the time of the subjects that used ETM.

Table 2 shows some of the data collected for EMeD. The first column shows the average time to complete tasks for each user group. We had (1) four knowledge engineers who had not used EMeD before but were familiar with EXPECT (2) two knowledge engineers not familiar with EXPECT, (3) four users not familiar with AI but had formal training in computer science, and (4) two users with no formal training in AI or CS. The second column shows the average number of problem-solving methods added. The last column shows the average time to build one problem solving method. The results for different user groups are shown separately to contrast the results. The last row in the table summarizes the results.

6.2.6. ASSESSING THE EVIDENCE FOR OR AGAINST HYPOTHESES AND CLAIMS

The following are our conclusions from the experiments with ETM:

- **Subjects using ETM (that made no mistakes) took less time** to complete the assigned KA tasks. These reductions on time depended on the complexity of the KA tasks and on the subject's previous experience with EXPECT. Subjects in both groups, ETM and the control group, made costly mistakes that severely affected their completion time yet handling those kinds of mistakes was outside of the scope of ETM. As a result, these data points are problematic. Section **Error! Reference source not found.** discusses this issue further. In (Tallis, 99) we present a detailed analysis of the cases that included mistakes and suggest that if the time incurred in handling the mistakes is subtracted then subjects in the control group take longer to complete the modification than subjects using ETM.

- The differences in time were **greater for the less experienced subjects.**

- The experiments **did not show clear evidence that ETM would reduce the number of user mistakes.**

90

- ETM was able to **guide subjects throughout all the required changes of the scenarios** for most of the subjects that used ETM. In the few cases in which subjects had to perform changes without ETM, these cases were out of the scope of ETM. This virtue of ETM was not predicted beforehand.

The following statements summarize our conclusions from the EMeD experiments:

- **Subjects using EMeD took less time** to complete the KA tasks. EMeD was able to reduce the development time to 2/3 of the time that users needed without it.

- The differences in time were not so evident for the less experienced subjects. The ratio for less experienced subjects remain about the same as the ratio for EXPECT users.

Additionally, the results show that subjects needed to add slightly less KB elements with EMeD. We may use this additional finding to explore some other hypotheses in the future, such as the effect of EMeD on the quality of the output KBS. Also, we may investigate why the ratios did not improve as predicted for the less experienced users. Sometimes the results provide only some evidence for our hypotheses because they may not be strong enough to confirm the hypotheses conclusively. This attests to the difficulty of designing this kind of experiments.

We also find very useful to analyze the data in detail, looking for interesting and unexpected phenomena. For example, the ETM experiment showed that subjects followed KA Scripts as checklists for the steps that needed to be performed. Consequently, the subjects that used ETM did not omit as many changes as the subjects in the control group, which was an interesting observation about why KA Scripts are useful.

6.3. Lessons Learned

To help others avoid some of the pitfalls that we encountered in the design of our experiments, we list here some of the issues that we had to address:

- **The experiment design is constrained by the limited number of users.** For many user studies of KA tools, the time required from subjects and experimenters and the specific qualifications that subjects have to meet (e.g., domain experts, knowledge engineers) constitute a practical limit in the number of subjects included in a study. This significantly impacts the design of an experiment. The experiment has to be carefully designed to control as much as possible the variables that affect the outcome of the experiment because the number of subjects involved tends to be small and we cannot rely in the

91

statistics to compensate for these variability.. Unfortunately, this variables are often very difficult to control.

- **User make mistakes that are outside the scope of the tool.** During the ETM experiments, subjects in both groups made (and then fixed) mistakes not related to the features of the tools that we were evaluating. The nature of these mistakes and the time that subjects spent fixing them varied. This was one of the factors that most severely affected the results of the experiments, in which some subjects spend more than half of their time interpreting and repairing their own mistakes. The following are some types of mistakes made by the subjects during the experiment that were not intended to be prevented by our KA tools:

  - Syntax errors: Subjects made syntax errors while entering new knowledge base elements using a text editor. Syntax errors were immediately detected by a parser included in both the basic and the enhanced versions of the KA tools and were reported back to the users. Although some errors were more difficult to interpret than others, all of these were simple to repair. Neither the text editor or the parser were among the KA tool features being evaluated.

  - Misuse of the domain structures: Subjects got confused while entering complex knowledge base elements that made reference to other elements of the KB. For example, a subject referred to the HEIGHT of a RIVER instead of to the HEIGHT of the BANK of a RIVER. Most of these errors were immediately detected by a KB verification facility included in both versions of the tool. These errors were reasonable simple to repair.

  - Misconceptions of the domain model: Subjects approached the KA task in a wrong way because they had misunderstood some aspects of the domain. For example, one subject wrongly believed that instance of SEAPORTS would point out to its LOCATION. However, this was not the relation that was represented in our model but its inverse (i.e., LOCATIONS pointed to its SEAPORTS instead of SEAPORTS to its LOCATIONS). Some of these errors were detected along the evolution of the KA task when the subjects encountered clear contradictions that made them revise their interpretation of the domain. Some other errors were not detected until the subjects, believing that they had finished the assigned task, checked the KB with the provided sample problems and obtained wrong results. Detecting and repairing these errors was very difficult and sometimes required to undo some modifications made erroneously by the user.

- Misunderstanding of the assigned KA task and/or omission of requested changes. Subjects performed a sequence of wrong modifications because they had misunderstood the assigned task or oversaw some changes. Locating and repairing these errors was ver difficult and sometimes required to undo some modifications made erroneously by the user.

- **Differences in subject performance.** In one of the experiments the individual differences in performance among subjects was so pronounced that it was very hard to compare results across subjects. To make things worse, there were no apparent indicators that could let us predict in advance the performance of subjects in order to assign them better to the different groups. This situation forced us to compare only the difference in performance in using both tools for each subject (i.e., to compare the performance of each subject in each one of the scenarios) and distorted the analyses based on aggregated data. The following are some of the characteristics that made subjects to perform differently: being extra cautious (e.g., they checked carefully each modification that the performed), exploring the tool's features during the experiment, critiquing the tool during the experiment, making decisions faster, typing faster or managing the text editor more skillfull .

- **Differences in understanding the domain, the assigned tasks, or the use of the KA tool.** Some subjects did not understand correctly the domain or the assigned task. This caused them to make mistakes or to stop in the middle of the experiment to clarify the instructions or the domain specifications. Other subjects did not understand some features of the KA tool and could not take advantage of them, which presumably made them take longer time to complete the tasks.

- **Different ways to solve the assigned tasks.** In some experiments, subjects solved the assigned tasks in different ways, hence the differences in performance are affected by the differences in the amount of work required to implement the different solutions. For example, some subjects defined few general KB elements that applied to several cases while others defined several specific KB elements that applied to few cases each; some subjects modified existing knowledge to handle new requirements while others added new knowledge to handle them; some subjects relied more in the tool's intrinsic inference capabilities while others preferred to state facts and procedures explicitly.

The design of experiments that control all of the above factors is not always feasible. Besides, there is always the possibility that other unforeseen factors also affect the outcome of the experiments. A practical alternative to enforcing stricter controls is to collect very fine grained measurements throughout the execution of the experiment and

based on these measurements analyze the results carefully. The collection of fine grained measurements has other advantages as well. The execution of a KA task involves the execution of several small activities. Table 3 lists some of the observed activities that subjects performed during the executions of the experiments. Usually, a KA tool supports only some of these activities. If we only take into account the subject overall performance we are also weighing some activities that are not related to our claims. In the future, we plan to isolate better the specific activities that our tools are intended to support.

We have learned from our experience that a careful experiment design can enhance the quality and utility of the collected fine gr ained measurements. For example, in one experiment we treated the edition of a KB elements (with a text editor) as a single KB modification action and we only recorded its initial and ending time. However, while editing a KB element and before closing the text editor, a subject might perform several activities which will not get individually recorded in our logs. For example, the subject might modify several different parts of that element, make a mistake and then fix it, and even spend some time deciding how to proceed with that modification. This deficiency in our instrumentation precluded us from isolating the time incurred in modifying individual aspects of the problem solving knowledge. This was unfortunate because we believed that the evaluated tool would be helpful only for some of these aspects and we could not corroborate our hypothesis.

The following list summarizes some of the lessons that we have learned from conducting our experiments.

- Use within subject experiments. Each subject should perform two tasks, one with the tool being evaluated and the other with the ablated tool. This helps to compensate for differences in user performance. Both tasks should be of comparable complexity.

- Minimize the variables unrelated to the claims to be proven. For example, in one of our experiments the KA tool allowed users to perform the same modification through different mechanisms: using a text editor or a menu based interface. The multiplicity of mechanisms to perform a modification did not add any value to the experiment, however it introduced unnecessary variability that complicated the analysis of the results.

- Minimize the chances that subjects make mistakes unrelated to the claims. Do not introduce unnecessary complications to the KA tasks. One of our experiments required that the subjects used a command that is hard to understand. Since the tool to be evaluated did not provide any support to handle this command it would have been better to avoid the need for that command in the experiment.

- Isolate as much as possible the KA activities and the data that are relevant to the hypotheses. It was a good decision for the evaluation of ETM to split the KA task in two parts: before and after the first KB modification because ETM is concerned with the latter. Discriminating these two parts helped to perform a more focused evaluation of ETM.

- Ensure that subjects understood the domain and the assigned KA task. In the EMeD experiments the subjects discussed with the experimenters their understanding of the assigned task. These subjects had less problems in executing the assigned tasks than the subjects in the ETM experiments.

- Avoid the use of text editors. The use of a text editor in our experiments caused subjects to make syntax errors. The differences in the subject's skills with the text editor program also affected the results of the experiments. It also did not allow us to discriminate the fine grained activities performed by the subjects.

6.4. A note on Statistical Analysis

As we mentioned earlier, the cost and resources required by empirical controlled user studies of KA tools result in relatively small scale experiments. Given the small number of subjects and tasks involved, it does not seem appropriate to analyze the statistical significance of our results. Researchers in other areas concerned with evaluation do not seem to consider this a crucial issue in current evaluation work. In any case, it is interesting to note that our results stand up to standard tests of statistical significance. For example, a t-test on the results reported in shows that they are significant at the 0.05 level with $t(2) = 7.03$, $p < .02$. Gathering data from more subjects within each group may be more reassuring than using these tests for validation.

6.5. Related Work on User Studies in Knowledge Acquisition and Other Fields

A few relevant evaluations of KA tools that have been conducted to date. We describe them in terms of the methodology that we have presented in this chapter. The studies are summarized highlighting the hypotheses/claims that were tested, the kinds of tasks and subjects used, the experimental setup, the results reported, and any findings that were surprising.

The TAQL study was done by Greg Yost as part of his PhD work at CMU. TAQL is a KA tool for SOAR. Yost evaluated the tool by itself and also evaluated its performance compared to some basic data that had been reported for two other KA tools (SALT and KNACK).

```
1) Evaluation of Taql
- Hypothesis: Taql has more breadth than other KA tools and
still effective
- KA task: implement a new KB given a domain description
```

- KB domains: 10 puzzles + 9 Expert systems
- Underlying KR: production rules
- Users: Soar programmers, three subjects (including. Yost)
- Experimental setup:
    - each subject given a domain description (domain-oriented, not
      implementation specs) + (at most) 3 test cases
    - three rounds of evaluations, starting with simple domains
- Data collected:
    - times for task understanding, design, coding, debugging
    - bug information: how found, what error, when and how fixed
- Results reported:
    - encoding rate (minutes per Soar production) for each subject
        in each domain
    - average fix time for catchable and uncatchable errors pre and post tool
- Conclusions:
    - subjects reduced their encoding rates over time
        (i.e., programmed faster)
    - encoding rate did not slow down as task size increased

2) Comparing Taql, Knack, and Salt
  - Users:
    - one subject for each case
    - reimplementation of original system (Knack and Salt cases)
  - Results reported:
    - development time (hours) for Taql and for two tools (Knack and Salt)
        at their respective domains (time reported for reimplementation)
  - Conclusions:
    - Taql outperformed role-limiting KA tools (this was a surprise)

The TURVY study was conducted by David Maulsby and Allen Cypher at the Advanced Technologies Group at Apple. This experiment was more on the area of HC rather than KA, but it is relevant here because it tests an approach to programming b demonstration that learns as a user performs simple tasks.

- Hypotheses:
    - H1: all users would employ same set of commands even
      if told nothing in advance about the instructions that

96

```
        Turvy understands,table of predicted set of commands was
        compiled in advance
        - H2: users would end up communicating using Turvy's
        terms
        - H4: users would tech Turvy simple tasks easily and
        complex tasks withreasonable effort
- KA tasks and KB domains:
        - modify bibliography format (main tests)
        - file selection
        - graphical editing
- Underlying KR: none
- Users: non-programmers
- Experimental setup:
        - "Wizard of Oz" experiment (no real software, user
        interacts with facilitator)
        - several rounds, different types of subjects
                - on main task:
                        - pre-pilot experiment
                        - pilot experiment (4 users)
                        - main experiment (8 subjects)
                - on other domains:
                        - 3 subjects
                        - 2 subjects
- Data collected:
        - videotapes, notes, interviews
- Results reported:
        - qualitative results mostly (their intention)
        - some quantitative results were obtained by post-
analysis
- Conclusions:
        - Evidence for H1, H2, H4
        - Interesting findings: quiet vs talkative users
```

There are other experiments in the field of KA that are not directly relevant but are worth mentioning. The Sisyphus experiments show how different groups would compare their approaches for the same given task, but most approaches lacked a KA tool and no user evaluations were conducted. A very controversial experiment tested whether knowledge engineering models (suc h as KADS models) were useful to users , but it tested knowledge elicitation through models and did not test any tools or systems. Other evaluations have tested the use and reuse of problem-solving methods, but they measure code reuse rather than how users benefit from KA tools.

Outside of KA, there are relevant studies in other disciplines. As we mentioned earlier, user evaluations are very uncommon in AI research. Most evaluations involve run -time behavior of AI software with no human in the loop . User studies are more common in software engineering, HCI, and intelligent tutoring systems.

In software engineering, empirical evaluations have been used for years to evaluate tools to support programmers. In this field, many aspects and issues in the software development process have been under study including languages, development environments, reuse, quality, and software management. User studies are only of concern for a few of these topics. Interestingly, the kind of controlled methods th at we report in this chapter generally seem to be in the minority when it comes to evaluate software. Many studies do not involve users, others analyze some historical data that may be available, and many collect observations and data as a software project unfolds without any particular control settings.

User studies in the field of HCI share many of the issues that arise in the evaluation of KA tools. An additional complication in evaluating interfaces is that they do not work in isolation, i.e., often times an interface can only be as good as the target system that the user ultimately operates on through the interface. On the other hand, many of the studies in this area can involve more users and settings, since the tasks tend to be simpler and the target users seem to be more numerous (e.g., users are not required to have domain expertise).

In intelligent tutoring systems, there are recognized tradeoffs regarding the merits and needs of different approaches to evaluation. Although formal eva luations are generall preferred, their cost makes them often impractical. Informal studies tend to be more common and seem to be sufficiently informative in practice to many researchers to guide their work.

Our studies to date do not address thoroughly the evaluation of the product itself, i.e., the knowledge base that results from the knowledge acquisition process. Currently, we test that the final knowledge base has sufficient knowledge to solve the right problems and generating the right answe rs. Other metrics, such as measures of the quality of the knowledge base, are also important. There is relevant work along these lines in the expert systems area as well as in software engineering. Our studies to date do not assess either how our particular tools would improve the end -to-end process of developing a knowledge base, which includes interviewing experts, building prototypes, maintaining the knowledge base, and improving system performance. There is relevant work in software engineering on evaluating the improvement to the overall software development process, including studies specific to expert systems as software.

6.6. Summary

We have presented a methodology for designing user evaluations of KA tools. We have been using it successfully in our own work to evaluate various approaches within the EXPECT framework. We have also discussed the lessons learned from our studies of two KA tools, and outlined some open issues. Our hope is that sharing our methodology and our experiences with the KA community we will contribute to making our field a more experimental and perhaps a more scientific one.

7. Evaluations with End Users

Developing tools that allow non-programmers to enter knowledge has been an ongoing challenge for AI. In r ecent years researchers have investigated a variety of promising approaches to knowledge acquisition (KA), but they have often been driven by the needs of knowledge engineers rather than by end users. This chapter reports on a series of experiments that we conducted in order to understand how far a particular KA tool that we are developing is from meeting the needs of end users, and to collect valuable feedback to motivate our future research. This KA tool, called EMeD, exploits Interdependency Models that relate individual components of the knowledge base in order to guide users in specifying problem -solving knowledge. We describe how our experiments helped us address several questions and hypotheses regarding the acquisition of problem-solving knowledge from end users and the benefits of Interdependency Models, and discuss what we learned in terms of improving not only our KA tools but also about KA research and experimental methodology.

7.1. Introduction

Acquiring knowledge from end users (i.e., ordinary users without formal training in computer science) remains a challenging area for AI researchinsertfootins Copyright á 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.. Many knowledge acquisition approaches target knowledge engineers and those that have been developed for end users only allow them to specify certain kinds of knowledge, i.e., domain -specific knowledge regarding instances and classes but not problem-solving knowledge about h ow to solve tasks. Alternative approaches apply learning and induction techniques to examples provided by users in a natural way as the are performing a task. Although these tools may be more accessible to end users, the are only useful in circumst ances where users can provide a variety of examples. When examples are not readily available, we may need knowledge acquisition (KA) tools for direct authoring.

In recent years, researchers have investigated a variety of new approaches to develop KA tools, in many cases targeted to end users though in practice motivated by knowledge engineers. Few user studies have been conducted and the participants are typically knowledge engineers. Without studies of the effectiveness of KA approaches and tools for end users, it is hard to assess the actual requirement of end users and our progress towards satisfying them. One of the challenges of this work is to devise a methodology and experimental procedure for conducting user studies of KA tools.

As KA researchers, we wanted to test our approach and KA tools with end users. A central theme of our KA research has been how KA tools can exploit *Interdependency Models*? that relate individual components of the knowledge base in order to develop expectations of what users need to add next. To give an example of interdependencies, suppose that the user is building a KBS for a configuration task that finds constraint violations and then applies fixes to them. When the user defines a new constraint, the KA tool has the expectation that the user should specify possible fixes, because there is

100

an interdependency between the problem-solving knowledge for finding fixes for violated constraints and the definitions of constraints and their possible fixes.

EMeD (EXPECT Method Developer) , a knowledge acquisition tool to acquire problem-solving knowledge, exploits Interdependency Models to guide users by helping them understand the relationships among the individual elements in the knowledge base. The expectations result from enforcing constraints in the knowledge representation system, working out incrementally the interdependencies among the different components of the KB. Our hypothesis is that Interdependency Models allow users to enter more knowledge faster, particularly for end users.

In addition to the goal of evaluating the role of Interdependency Models, we had more general questions. Users with different degrees of exposure to computing environments would probably perform differently. But in what ways? How much training and of what kind is needed before they can make reasonably complex additions to a knowledge base with a KA tool? What aspects of a knowledge base modification task are more challenging to end users? What kinds of interfaces and interaction modalities would be appropriate and in what ways should they be different from those that knowledge engineers find useful

This chapter reports on a study to evaluate our KA tools with domain experts (end users) who extended a knowledge base in their area of expertise. This study was conducted as part of an evaluation of the DARPA High Performance Knowledge Bases program. We also present our experimental design and the preliminary study with users with varying degrees of background in AI and computer science, which was performed before the evaluation. We analyze the results in terms of our initial questions and hypotheses, and extract some general conclusions that motivate future directions of KA research.

7.2. EMeD: Exploiting Interdependency Models to Acquire Problem-Solving Knowledge

```
((name method1)
 (capability (check (obj (?f is (spec-of force-ratio)))
                    (of (?t is (spec-of main-task)))
                    (in (?c is (inst-of COA)))))
 (result-type (inst-of yes-no))
 (method (check (obj (spec-of force-ratio))
                (of (main-task-of (close-statement-of ?c)))))
((name method2)
 (capability (check (obj (spec-of (force-ratio)))
                    (of (?t is (inst-of military-task)))))
 (result-type (inst-of yes-no))
 (method (is-less-or-equal
              (obj (estimate (obj (spec-of required-force-ratio))
                             (for ?t)))
              (than (estimate (obj (spec-of available-force-ratio))
                              (for ?t)))))))
((name method3)
 (capability (estimate (obj (?f is (spec-of required-force-ratio)))
                       (for (?s is (inst-of military-task)))))
 (result-type (inst-of number))
 (method ...))
((name method4)
 (capability (estimate (obj (?f is (spec-of available-force-ratio)))
                       (for (?t is (inst-of military-task)))))
 (result-type (inst-of number))
 (method  ))
```

Table 3: Examples of EXPECT Problem-Solving Methods.

EMeD (EXPECT Method Developer) is a knowledge acquisition tool that allows users to specify problem-solving knowledge. This section summarizes the functionality of the tool, further details and comparison with other tools are provided in.

EMeD is built within the EXPECT framework. EXPECT's knowledge base contains ontologies that describe the objects in a domain, and problem-solving methods that describe how tasks are achieved. Tasks are specified as goal hierarchies, where a goal is broken into smaller subgoals all the way down to primitive or basic tasks. The problem-solving methods specify how the decomposition takes place. EXPECT provides a rich language that was developed with understandability and intelligibility in mind, since it was used to generate adequate explanations for knowledge-based systems. Figure 1 shows some examples of EXPECT methods. Each problem-solving method has a *capability* that describes what the method can achieve, a *result type* that specifies the kind of result that the method will return upon invocation, and a *method body* that specifies the procedure to achieve the capability. The method body includes constructs for invoking subgoals to be resolved with other methods, retrieving values of concept roles, and control constructs such as conditional expressions and iteration. The arrows in

the figure indicate some interdependencies, where a head of an arrow points to a sub-method which can solve a given subgoal. For example, the second method shown in the Figure 1 checks the force ratio of a given military task by comparing its required force ratio and the available force ratio. The result should be yes or no depending on whether the required ratio is less than the available ratio.

EXPECT derives an *Interdependency Model* (IM) by analyzing how individual components of a knowledge base are related and interact when they are used to solve a task. An example of interdependency between two methods is that one may be used b the other one to achieve a subgoal in its method body. Two methods can also be related because they have similar capabilities. EMeD exploits IM in three ways: (1) pointing out missing pieces at a given time; (2) predicting what pieces are related and how; (3) detecting inconsistencies among the definitions of the various elements in the knowledge base.
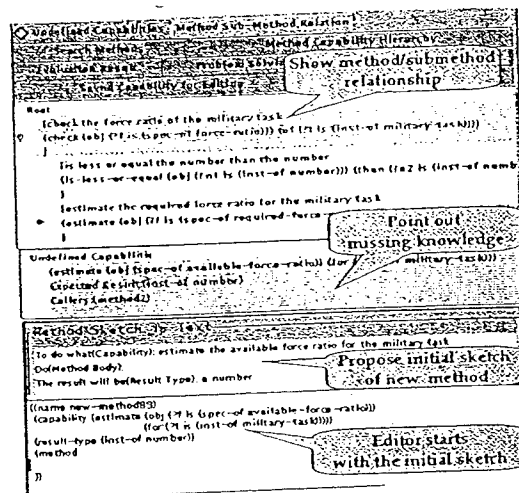

Table 4: The Method Proposer of the EMeD Acquisition Interface.

When users define a new problem-solving method, EMeD first finds the interdependencies and inconsistencies within that element, such as if any undefined variable is used in the body of the method. If there are any errors within a method definition, the *Local-Error Detector* displays the errors and it also highlights the incorrect definitions so that the user can be alerted promptly. The *Global-Error Detector* analyzes the knowledge base further and detects more subtle errors that occur in the context of problem solving.

By keeping the interdependencies among the problem-solving methods and factual knowledge, and analyzing interdependencies between each method and its sub-methods,

the *Method Sub-method Analyzer* in EMeD can detect missing links and can find undefined problem-solving methods that need to be added. EMeD highlights those missing parts and proposes an initial version of the new methods, as shown in Figure 2 In this example, a method for checking the force ratio for an assigned task needs to compare the available force ratio (i.e, ratio between blue units and red units) with the force ratio required for that task. When the system is missing the knowledge for the available ratio (i.e., missing method4), the *Method Proposer* in EMeD notifies the user with a red diamond (a diamond shown in Figure 2 on the top) and displays the ones needed to be defined. It can also construct an initial sketch of the capability and the result type of the new method to be defined. What the new method has to do (capability of the method) is to estimate the available force ratio for a given military task. Since we are computing a ratio, the result type suggested is a number (method sketch in Figure 2). Users can search for existing methods that can achieve a given kind of capability using the *Method-Capability Hierarchy*, a hierarchy of method capabilities based on subsumption relations of their goal names and their parameters.

Finally, EMeD can propose how the methods can be put together. By using the Method Sub-method Analyzer for analyzing the interdependencies among the KB elements, it can detect still unused problem-solving methods and propose how they may be potentially used in the system.

7.3. Experimental Design

As described in the introduction, current KA research lacks evaluation methodology. In recognition of the need for evaluation, the community started to design a set of standard task domains that different groups would implement and use to compare their work. These Sisyphus experiments show how different groups would compare their approaches for the same given task, but most approaches lacked a KA tool and no user evaluations were conducted. Other evaluations have tested the use and reuse of problem-solving methods, but they measure code reuse rather than how users benefit from KA tools. Other KA work evaluated the tool itself. TAQL's performance was evaluated b comparing it with some basic data that had been reported for other KA tools. There were some user studies on ontology editors. In contrast with our work, these evaluations were done with knowledge engineers. Also since the experiments were not controlled studies, the results could not be causally linked to the features in the tools. Our research group has conducted some of the few user studies to date, and as a result we have proposed a methodology that we use in our own work. It turns our that the lack of user studies is not uncommon in the software sciences. In developing a methodology for evaluation of KA tools, we continue to draw from the experiences in other areas.
Our goal was to test two main hypotheses, both concerned with Interdependency Models (IMs):

Hypothesis I: A KA tool that exploits IMs*enables users to make a wider range of changes* to a knowledge base because without the guidance provided with IMs users will be unable to understand how the new knowledge fits with the existing knowledg and complete the modification.

Hypothesis II: A KA tool that exploits IMs *enables users to enter knowledge faster* because it can use the IMs to point out to the user at any given time what additional knowledge still needs to be provided.

There are three important features of our experiment design:

- In order to collect data comparable across users and tasks, we used a controlled experiment. Thus, we designed modification tasks to be given to the participants based on typical tasks that we encountered ourselves as we developed the initial knowledge base.

- Given the hypotheses, we needed to collect and compare data about how users would perform these tasks under two conditions: with a tool that exploits IMs and with a tool that does not (this would be the control group). It is very important that the use of IMs be the only difference between both conditions. We designed an ablated version of EMeD that presented the same EMeD interface but did not provide any of the assistance based on IMs.

- Typically, there are severe resource constraints in terms of how many users are available to do the experiments (it typically takes several sessions over a period of days). In order to minimize the effect of individual differences given the small number of subjects, we performed within-subject experiments. Each subject performed two different but comparable sets of tasks (each involving the same kind of KA tasks but using a different part of the knowledge base), one with each version of the tool.

In order to determine when a KA task was completed, the subjects were asked to solve some problems and examine the output to make sure they obtained the expected results. In addition, after each experiment, we checked by hand the knowledge added by the subjects.

Participants were given different combinations of tools and tasks and in different order, so as to minimize transfer effects (i.e., where they would remember how they did something the second time around).

EMeD was instrumented to collect data about the user's performance, including actions in the interface (e.g., commands invoked and buttons selected), the knowledge base contents at each point in time, and the time at which each user action takes place. These provide objective measurements about task completion time and the use of specific features. Since this data was insufficient to understand what things users found hard and difficult to do with the tool or why a certain action was not taken, we collected additional information during the experiment. We asked users to voice what they were thinking and what they were doing and recorded them in transcripts and in videotapes (during the experiments with domain experts). We also prepared a questionnaire to get their feedback, where instead of questions with free form answers we designed questions that could be answered with a grade from 1 (worst) to 5 (best).

7.4. Preliminary Study

Since it is expensive to run user studies and hard to get domain experts in the field, we wanted to filter out distractions which are unrelated with our claim, such as problems with the tool that are not related to Interdependency Models. We also wanted to understand whether our interface and KA tool are appropriate for end users and how different types of users interact with it, so that we can improve our tools and our experimental methodology. For these reasons, we performed a preliminary study before the actual evaluation with domain experts.

The study used a spectrum of users that had gradually less background in AI and CS. We had (1) four knowledge engineers who had not used EMeD before but were familiar with EXPECT, (2) two knowledge engineers not familiar with EXPECT but that had experience with knowledge-based systems, (3) four users not familiar with AI but had formal training in computer science, and (4) two users with no formal training in AI or CS.
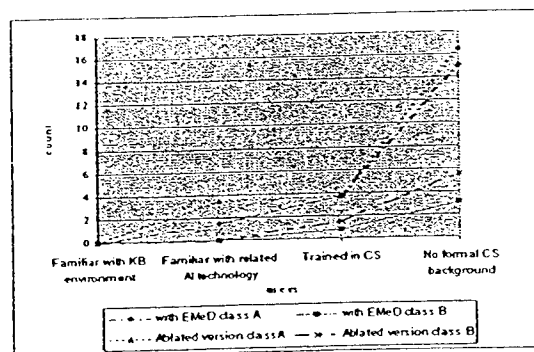
Table 5: Average number of hints given to each group of subjects during the preliminary user study.

Since a major goal of this preliminary study was to understand our KA tool, we allowed the subjects to ask for hints when they were not able to make progress in the task (this was not allowed in the final evaluation). These hints allow us to categorize the basic types of difficulties experienced by users and adjust the tool based on them.
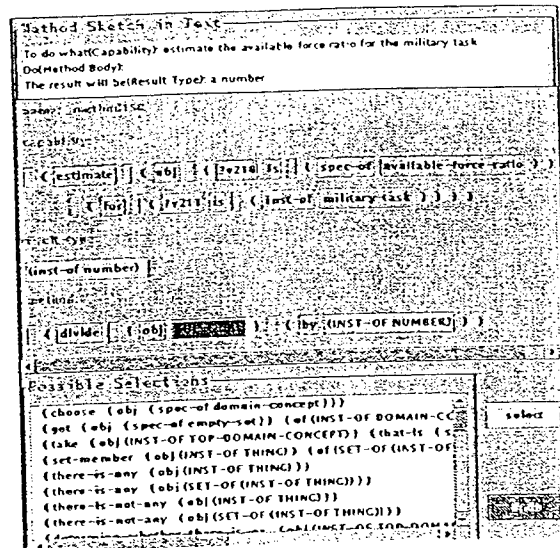
Table 6: Structured Editor.

Figure 3 shows the number of hints given to the subjects in this stud . More hints were always needed with the ablated version. The number of hints increases dramatically when subjects lack CS background. We analyzed all the hints, and separated them into two major categories. Class A hints consist of simple help on language and syntax, or clarification of the tasks given. Since syntax errors are unrelated to our claims about IMs, we developed a Structured Editor for the new version of EMeD (version 2) that guides users to follow the correct syntax. Figure 4 shows the new editor which guides the users to follow the correct syntax. Users can build a method just using point and click operations without typing.

Class B hints were of a more serious nature. For example, users asked for help to compose goal descriptions, or to invoke a method with the appropriate parameters. Although the number of times these hints were given is smaller and the number is even smaller with EMeD, they suggest new functionality that future versions of EMeD should eventually provide to users. The subjects indicated that sometimes the tool was showing too many items, making it hard to read although they expected this would not be a problem after they had used the tool for a while and had become used to it. Since these presentation issues were affecting the results of the experiment and are not directl evaluating the IMs, the new version of EMeD (version 2) has more succinct views of some of the information, showing details only when the user asks for them. Other hints pointed out new ways to exploit IMs in order to guide users and would require more substantial extensions to EMeD that we did not add to the new version. One area of difficulty for subjects was expressing composite relations (e.g., given a military task,

retrieve its assigned units and then retrieve the echelons of those assigned units). Although EMeD helped users in various ways to match goals and methods, in some cases the users still asked the experimenters for hints and could have benefited from additional help. The fundamental difficulties of goal composition and using relations still remained as questions for the real experiment.

In addition to improving the tool, we debugged and examined our experimental procedure, including tutorial, instrumentation, questionnaire, etc., especially based on the the results from the fourth group.

We found out how much time end users would need to learn to use our tools. The tutorial given to the users was done with simpler sample tasks from the same knowledge base. The training time was significantly longer and harder for the subjects with no technical background (2 hours for knowledge engineers and 7.5 hours for the project assistants). More details of this study are discussed in, showing that even the end users were able to finish complex tasks, and that the KA tool saves more time as users have less technical background.

As described above, we extended our tool based on the pre-test results, creating a new version of EMeD (version 2). The next section describes the evaluation with domain experts with this new version of EMeD.

7.5 Experiment with Domain Experts

The participants in this experiment were Army officers facilitated by the Army Battle Command Battle Lab (BCBL) at Ft Leavenworth, KS. They were asked to use our KA tools to extend a knowledge based system for critiquing military courses of action. Each subject participated in four half-day sessions over a period of two days. The first session was a tutorial of EXPECT and an overview of the COA critiquer. The second session was a tutorial of EMeD and a hands-on practice with EMeD and with the ablated version. In the third and fourth sessions we performed the experiment, where the subjects were asked to perform the modification tasks, in one session using EMeD and in the other using the ablated version. Only four subjects agreed to participate in our experiment, due to the time commitment required.

An important difference with the previous study is that during this experiment subjects were not allowed to ask for hints, only clarifications on the instructions provided. As soon as a participant would indicate that they could not figure out how to proceed, we would terminate that part of the experiment. In order to collect finer-grained data about how many tasks they could complete, we gave each subject four knowledge base modification tasks to do with each version of the KA tool. The reason is that if we gave them one single task and they completed almost but not all of it then we would not have any objective data concerning our two initial hypotheses. The four tasks were related, two of them were simpler and two more complex. The easier tasks required simple modifications to an existing method (e.g., generalize the existing methods that compute the required force ratio for "destroy" tasks into methods that can compute the ratio for

any military tasks in general). The more complex tasks required adding new methods, such as the second method shown in Figure 1.

7.6. Results and Discussion

The main results are shown in Figure 5. Figure 5-(a) shows the average time to complete tasks (for the completed tasks only). None of the subjects was able to do the more complex tasks with the ablated version of EMeD. Where data is available (the easier tasks), subjects were able to finish the tasks faster with EMeD. Figure 5-(b) shows the number of tasks that the subjects completed with EMeD and with the ablated version, both by task category and overall. The solid part of the bars show the number of tasks completed. We show with patterned bars the portion of the uncompleted tasks that was done when the subjects stopped and gave up (we estimated this based on the portion of the new knowledge that was added). Figure 5-(c) shows the same data but broken down by subject [6]. The results show that on average subjects were able to accomplish with EMeD almost twice as many tasks as they accomplished with the ablated version. The results support our claims that Interdependency Models can provide significant help to end users in extending knowledge bases.
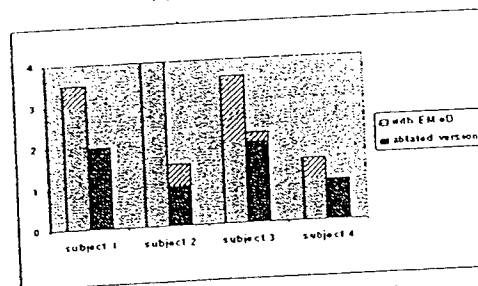
---

[6]We had noticed early on that Subject 4 had a different background from the other three, but unfortunately we were not able to get an alternative subject.
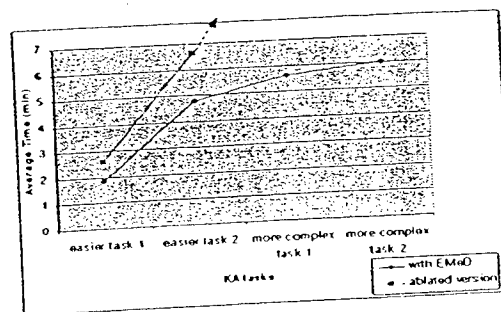
(a) Average time to complete tasks
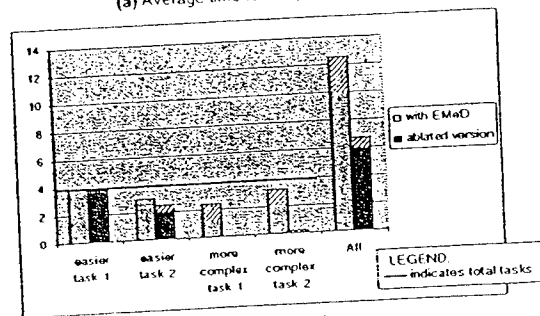
(b) Tasks completed

(c) Tasks completed (for each subject)

Figure 7: Results of the evaluation with domain experts.

It would be preferable to test additional subjects, but it is often hard for people (especially domain experts) to commit the time required to participate in this kind of study. Given the small number of subjects and tasks involved it does not seem appropriate to analyze the statistical significance of our results, although we have done so for some of the initial experiments with EMeD with a t-test showing that they were significant at the 0.05 level with $t(2)=7.03$, $p < .02$. Gathering data from more subjects may be more reassuring than using these tests for validation.

Our experience with these experiments motivates us to share a few of the lessons that we have learned about knowledge acquisition research:

☐ **Can end users use current KA tools to modify the problem-solving knowledge of a knowledge-based system? How much training do they need to start using such KA tools? Would they be able to understand and use a formal language?**

As we described earlier, we spent 8 hours (two half-day sessions) for training. They spent roughly half of that time learning EXPECT's language and how to put the problem-solving methods together to solve problems. The rest of the time was spent learning about the KA tool and its ablated version. We believe that this time can be reduced by improving the tool's interface and adding on-line help. We also recognize that more training may be needed if users are expected to make much more complex changes to the knowledge base. At the same time, if they did not need to be trained on how to use an ablated version of the tool they would not need to learn as many details as our subjects did.

Our subjects got used to the language and could quickly formulate new problem-solving methods correctly. They did not seem to have a problem using some of the complex aspects of our language, such as the control structures (e.g., if-then-else statement) and variables. It took several examples to learn to express procedural knowledge into methods and sub-methods and to solve problems. EMeD helps this process by automatically constructing sub-method sketches and showing the interdependencies among the methods. Retrieving role values through composite relations was also hard. Providing a better way to visualize and to find this kind of information would be very useful.

As a result of this experiment, we believe that with current technology it is possible to develop KA tools that enable end users to add relatively small amounts of new problem solving knowledge, and that they can be trained to do so in less than a day.

☐ **How much do Interdependency Models help? What additional features should be added to our KA tools?**

Functionalit	Avg No. invocations	Usefulness rating	No Users who used it
Method Proposer	10.5 (1.25	4.7	4
Method Sub-method Analyzer	8.5	4.3	4
Method-Capabilit Hierarchy	2.75	4.5	2
Global-error Detector	3	3.3	4

Table 1: Average Use of EMeD's Functionality.

Overall, the Interdependency Models exploited via different features in EMeD were useful for performing KA tasks. Table 1 shows the average use of each of the Components of EMeD, in terms of the number of times the user invoked

them [7]. The subjects were very enthusiastic about the tool's capabilities, and on occasion would point out how some of the features would have helped when they were using only the ablated version.

According to the answers to the questionnaire, using EMeD it was easier to see what pieces are interrelated. That is, visualizing super/sub method relations using Method Sub-method Analyzer was rated as useful (4.3/5). Also detecting missing knowledge and adding it was easier with EMeD's hints. Highlighting missing problem-solving methods and creating initial sketch based on interdependencies (by Method Proposer) were found to be the most useful (4.7/5).

The Structured Editor used in this version of EMeD provided very useful guidance, and there were less errors for individual method definitions. The Local-Error Detector was not used for the given tasks.

□ **What aspects of a modification task are more challenging to end users?**
Almost everyone could do simple modifications, which required that the subjects browse and understand the given methods to find one method to be modified and then changing it.
Some subjects had difficulties starting the KA tasks, when EMeD does not point to a particular element of the KB to start with. Although they could use the search capability in EMeD or look up related methods in the Method-Capability Hierarchy, this was more difficult for them than when the tool highlighted relevant information.
Typically, a KA task involves more than one step, and sometimes subjects are not sure if they are on the right track even if they have been making progress. A KA tool that keeps track of what they are doing in the context of the overall task and lets them know about their progress would be very helpful. Some of the research in using Knowledge Acquisition Scripts to keep track of how individual modifications contribute to complex changes could be integrated with EMeD.

□ **How do KA tools need to be different for different kinds of users**
We did not know whether end users would need a completely different interface altogether. It seems that a few improvements to the presentation in order to make the tool easier to use was all they needed. We did not expect that syntax errors would be so problematic, and developing a structured editor solved this problem easily. On the other hand, we were surprised that end users found some of the features useful when we had expected that they would cause confusion. For example, a feature in the original EMeD that we thought would be

---

[7]We show the number of times the users selected them, except for the Method Proposer where we show the number of times the system showed it automatically as well as the number of times selected (in parenthesis) when applicable.

distractive and disabled is organizing problem-solving methods into a hierarchy. However, the feedback from the end users indicates that they would have found it useful.

Although EMeD is pro-active in providing guidance, we believe that some users would perform better if we used better visual cues or pop-up windows to show the guidance. As the users are more removed from the details, the KA tool needs to do a better job at emphasizing and making them aware of what is important

7.7. Summary

In this chapter, we presented an evaluation of a KA tool for acquiring problem-solving knowledge from end users who do not have programming skills. We described the experimental procedure we have designed to evaluate KA tools, and how we refined the design with a preliminary user study with users with gradually less background in AI and computer science. The KA tool that we tested exploits Interdependency Models, and the results show that it helped end users to enter more knowledge faster. We also discussed additional lessons that we have learned that should be useful to other knowledge acquisition researchers.

Bibliography

[Aamodt et. al., 1993] Aamodt, A. et. al. Task features and their use in CommonKADS. Technical report KADS-II/T1.5VUB/TR/014/1.0, Free University of Brussels, 1993.

[Arens et al. 1996] Arens, Y., Knoblock, C., and Shen, w. Query Reformulation for Dynamic Information Integration. In Journal of Intelligent Information Systems, 1996.

[Bennett 1985] Bennett, J. S. ROGET: A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system. In Journal of Automated Reasoning, 1, pp. 49-74, 1985.

[Breuker and van de Velde 1994] Breuker, J. and van de Velde, W. CommonKADS Library for Expertise Modelling. IOS Press, Amsterdam, 1994.

[Chandrasekaran, 1986] B. Chandrasekaran. Generic tasks in knowledge -based reasoning. IEEE Expert , 1(3):23-30, 1986.

[Clancey 1985] Clancey, W.J., Heuristic classification. Artificial Intelligence, 27(3):289-350, 1985.

[Eshelman 1988] Eshelman, L. MOLE: A knowledge-acquisition system for cover-and-differentiate systems. In S. Marcus (Ed.), Automating Knowledge Acquisition for Expert Systems, Kluwer Academic Publishers, Boston, 1988.

[Gil et al. 1994] Gil, Y., Hoffman, M., and Tate, A. Domain -specific criteria to direct and evaluate planning systems. In ARPA/Rome Laboratory Knowledge -based Planning and Scheduling Initiative Workshop Proceedings, pp. 433-444, Tucson, Arizona, February 1994.

[Gil and Paris 1994] Gil, Y., and Paris, C.L. Towards Method-Independent Knowledge Acquisition. In Knowledge Acquisition, 6 (2), pp. 163-178, 1994.

[Gil and Swartout 1994] Gil, Y. and Swartout, W. EXPECT: a Reflective Architecture for Knowledge Acquisition. In ARPA/Rome Laboratory Knowledge-based Planning and Scheduling Initiative Workshop Proceedings, pp. 433 -444, Tucson, Arizona, February 1994.

[Gil 1994] Gil, Y. Knowledge Refinement in a Reflective Architecture. In Proceedings of the National Conference on Artificial Intelligence (AAAI-94), 1994.

[Gil and Melz 1996] Gil, Y. and Melz, E. Explicit Representations of Problem -Solving Methods for Knowledge Acquisition. In Proceedings of the Na tional Conference on Artificial Intelligence (AAAI-96), 1996.

[Klinker et al. 1991] Klinker, G., Bhola, C., Dallemagne, G., Marques, D., and McDermott, J. Usable and reusable programming constructs, In Knowledge Acquisition, 3 (2), pp. 117-135, 1991.

[Langley and Simon 1995]. Langley, P. and Simon, H. A. Applications of Machine Learning and Rule Induction. Communications of the ACM, 38(11) 1995.

[MacGregor 1988] MacGregor, R. A Deductive Pattern Matcher. In Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88). St. Paul, MN, August 1988.

[MacGregor 1990] MacGregor, R. The Evolving Technology of Classification-Based Knowledge Representation Systems. In John Sowa (ed.), Principles of Semantic Networks: Explorations in the Representation of Knowledge. Morgan Kaufmann, 1990.

[MacGregor 1994] MacGregor, R. A Description Classifier for the Predicate Calculus. In Proceedings of the Twelfth National Conference on Artificial Intelligence, (AAAI -94), pp. 213-220, 1994.

[Marcus and McDermott 1989] Marcus, S., and McDermott, J. SALT: A knowledge acquisition language for propose-and-revise systems. In Artificial Intelligence 39 (1), pp. 1-37, 1989.

[McDermott 1988] McDermott, J, Preliminary steps toward a taxonomy of problem solving methods. In Automating Knowledge-acquisition for expert systems S.Marcus (ed), Kluwer Academic Publishing, 1988.

[Mitchell et al. 1983]. Mitchell, T., Utgoff, P., and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem -Solving Heuristics. In "Machine Learning: An Artificial Intelligence Approach, Volume I, R. Michalski, J. Carbonell, and T. Mitchell (Eds), Tioga, 1983.

[Moore and Paris 1993] Moore, J. D., and Paris, C. L. Planning text for advisory dialogues: Capturing intentional and rhetorical information. In Computational Linguistics, 19 (4), 1993.

[Moore and Swartout 1989] Moore, J. D., and W. R. Swartout. A reactive approach to explanations. In Proceedings of the Eleventh International Conference on Artificial Intelligence, pp. 1505-1510, Detroit, Michigan, August 1989.

[Moore and Swartout 1990] Moore, J. D., and Swartout, W. R. Pointing: A way towards explanation dialogue. In Proceedings of the Eighth National Conference on Artificial Intelligence, pp. 457-464, Boston, Massachusetts, August 1990.

[Musen 1992] Musen, M. A. Overcoming the limitations of role -limiting methods. In Knowledge Acquisition 4 (2), pp. 165-170, 1992.

[Musen and Tu 1993] Musen, M. A., and Tu, S. W. Problem -solving models for generation of task-specific knowledge acquisition tools. In J. Cuena (Ed.), Knowledge -Oriented Software Design, Elsevier, Amsterdam, 1993.

[Porter et al. 1990]. Porter, B., Bareiss, R., and Holte, R. Concept Learning and Heuristic Classification in Weak-Theory Domains. Artificial Intelligence 45, pp229-263, 1990.

[Schreiber et al. 1993] Schreiber, A., Wielinga, B. and Breuker, J. KADS: A Principled Approach to Knowledge-Based Development. Academic Press, London, 1993.

[Swartout and Gil 1995] Swartout, W.R. and Gil , Y. EXPECT: Explicit Representations for Flexible Acquisition in Proceedings of the Ninth Knowledge Acquisition for

Knowledge-Based Systems Workshop (KAW'95) Banff, Canada, February 26-March 3, 1995

[Swartout and Moore 1993] Swartout, W. R., and Moore, J. D. Explanation in second-generation expert systems. In J.-M. David, J.-P. Krivine, and R. Simmons (Eds.), Second Generation Expert Systems, Springer-Verlag, 1993.

[Swartout et al. 1991] Swartout, W.R., Paris, C.L., and Moore, J.D. Design for Explainable Expert Systems, In IEEE Expert 6 (3), pp. 58–64, June 1991.

[Valente et al. 1994] Valente, A., van de Velde, W. and Breuker, J. CommonKADS Library for Expertise Modelling. In CommonKADS Expertise Modeling Library, Chapter 3, pages 31-56, J. Breuker and W. van de Velde, editors. IOS Press, Amsterdam, 1994.

[Valente et al. 1996] Valente, A., Gil, Y. and Swartout, W. R. INSPECT: an Intelligent System for Air Campaign Plan Evaluation based on EXPECT. ISI Technical memo, June 1996. Available on the Web at : the URL: http://www.isi.edu/~valente/inspect/inspect.html.

[Wilkins 1988] Wilkins, D. E. Practical Planning: Extending the Classical AI Planning Paradigm, Morgan Kaufmann, 1988.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.